1997

A class-based approach to parallelization of legacy codes

Simanta Mitra
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A class-based approach to parallelization of legacy codes

by

Simanta Mitra

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Major Professor: Dr. Suraj Kothari

Iowa State University

Ames, Iowa

1997

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Simanta Mitra

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College

TABLE OF CONTENTS

ACKNOWLEDGEMENT

My deepest appreciation to my wife, Soma, for her unwavering support and understanding during my graduate studies. Her cheerfulness in both hard times and good times was a constant source of inspiration.

My parent's and sister's contribution toward my education is immeasurable. In particular, I am deeply indebted to my mother for her guidance, encouragement, and belief in me.

I am very grateful to Dr. Suraj Kothari for teaching me to ask the right questions, for his guidance in my studies, and for being available and willing to discuss academic issues at any time of the day or night.

Special thanks go to Dr. Kelvin Nilsen, Dr. Frank Rizzo, and Dr. Arne Hallam. Dr. Nilsen taught me the compiler techniques without which it would not have been possible for me to develop parAgent. The preliminary work on BEM with Dr. Rizzo proved crucial in our understanding of parallelization of legacy codes. Dr. Hallam and the Department of Economics supported me for most of my years at ISU.

I thank Dr. John Gustafson, Dr. Gurpur Prabhu, Dr. Satish Udpa, Dr. Frank Rizzo, and Dr. Ambar Mitra for their helpful comments and suggestions towards the completion of this dissertation.

Appreciation is extended to my colleague, Dr. Youngtae Kim, for the many interesting discussions on MM5.

I am indebted to the many teachers I have encountered over the years for inspiring and educating me. I thank Dr. B. Patra, Dr. A. Sinha, and Dr. A. M. Ghosh for helping me to pursue higher education.

Finally, I acknowledge the role of my son, Akash, in inspiring me to finally complete this thesis.

CHAPTER 1 INTRODUCTION

1.1 Background

The vast majority of scientific and engineering studies are based on numerical modeling of physical phenomena. The computation-intensive legacy codes (valuable codes like MM5 which have been around for decades, usually written in Fortran-77) for these numerical models stand to benefit from application of parallel computing. Environmental modeling provides a good example. Numerous independent models dealing with segregated aspects of the environment, have evolved to a high level of sophistication. A serious impediment to creating combined comprehensive multi-process, multi-media ecological models is their computational demand: they span numerous physical and temporal scales which require large computational loops over space and time variables. This computational demand can be met by use of parallel computing.

The technological infrastructure for parallelization must make use of the existing codes. The solid pedigree associated with legacy codes make them very valuable. Typically, these codes have evolved over the years and gone through many refinement phases to make them robust and comprehensive for the study of a particular phenomenon. For instance, the Penn. State/ NCAR MM5, the widely used fifth generation Mesoscale meteorology model [4], represents almost two decades of development efforts. As the example of MM5 shows, scientists have invested enormous time and effort into developing such codes and therefore the issue of reuse of software is vital with respect to legacy codes.

The goal is to facilitate *quick* development of *efficient parallel* versions for *legacy* codes.

1.2 The Problem

There are, however, major hurdles in application of parallel computing to these legacy codes. These codes are very large and very complex. MM5 illustrates this well; it consists of more than 200 files, several hundred variables, and about one hundred thousand lines of

code. Multiple factors including the physics of the problem, the mathematical model, and the particular numerical technique for solving the problem contribute to the complex semantics.

The magnitude of effort required for manual parallelization is prohibitively large. It is very time consuming to manually parallelize (transform from sequential program to parallel program) these codes. The manual approach is prone to errors and debugging some of those errors can be very difficult to trace. For example, the parallelization of the non-hydrostatic version of MM5 [27] took about three and half years for a team of scientists at Argonne National Laboratory (ANL).

Difficulties in manual parallelization point to a need for automation. Over the years, scientists have developed several automatic and semi-automatic tools [11][25]. Doreen Cheng has published an extensive survey [8] with 94 entries for parallel programming tools out of which 9 are identified as "parallelization tools to assist in converting a sequential program to a parallel program." In spite of considerable efforts, attempts to develop fully automatic parallelization tools have not succeeded. Consequently, the emphasis of recent research has been on developing interactive (i.e. requiring assistance from the user) tools. Interactive D-editor[20] and Forge [16] are examples of state-of-the-art interactive parallelization tools. However, they too have a number of weaknesses and limitations and are unable to successfully parallelize legacy codes.

The situation is that while a large number of parallelization tools exist, quick development of efficient parallel codes for existing legacy codes is still not feasible.

1.3 Prior Experience

Before our attention was directed to this problem, we spent several years working on parallel computers. Our initial efforts were directed towards development of efficient parallel codes for various linear algebra routines. First, we developed and analyzed several matrix multiplication codes on MIMD computers. Performance studies were undertaken for the Cannon's, Systolic, Broadcast, and Straussen's algorithms for matrix multiplication. This was followed by performance study of LUD and FFT algorithms and development of efficient codes for sparse Cholesky algorithm.

After these initial studies, we undertook the parallelization of a BEM code. This was a large code and we spent a considerable amount of time understanding the semantics of the

code and also understanding the BEM method. During this time we developed an understanding and appreciation for the BEM, FEM, and FD techniques.

At this time, the manual parallelization of MM5, an explicit FD code, came to our attention. From our understanding of FD codes, it occurred to us that the process could probably be automated and we developed several tools to extract key information from the serial codes.

1.4 The Solution

Based on our experiences with parallelization of legacy codes, we have developed a new approach to automatic parallelization [29]. In this approach, automatic parallelization is focused on key classes. For example, consider the following numerical methods: the finite difference method (FDM), the finite element method (FEM), and the boundary element method (BEM). The vast majority of scientific and engineering codes are primarily based on these three numerical methods and if automatic parallelization can handle these classes, a broad spectrum of scientific and engineering applications will benefit.

Historically, a similar shift in focus has occurred in the domain of artificial intelligence when expert systems were introduced. While the problem of developing an intelligent program is too difficult in general, the idea of expert systems has proven to be fruitful in addressing important problems. The advantage of focusing on a class of problems is that the class-specific high-level knowledge can be used to simplify the otherwise intractable problem of parallelization. In our approach, the system requires the user to specify the high-level knowledge, but it automates tasks which are time-consuming, tedious, and error-prone for the user. The automatic analysis and transformations of the sequential code to arrive at the parallel code are based on the high-level knowledge of the numerical method.

This approach overcomes the hurdles of parallelization and facilitates quick development of efficient parallel codes for legacy codes. Using our approach, we have developed parAgent - a parallelizing tool which facilitates quick development of efficient parallel codes for legacy Fortran-77 codes based on 3 dimensional time-marching explicit finite difference model.

parAgent and the new approach are described in the following chapters. Chapter 2 discusses existing research on the parallelization problem. Chapter 3 presents the key

concepts in our approach. parAgent is described in detail in Chapter 4. In Chapter 5, the capabilities of parAgent are explained by demonstration on test cases. Summary statements are presented in Chapter 6.

CHAPTER 2 LITERATURE REVIEW

There exists a large body of work on parallelization techniques and tools. Wolfe [43] contains a wealth of references to research on parallel compiler techniques, Valiant [39], Culler [9], and Synder [37] discuss programming models for parallel computing, and Cheng's survey [8] provides a concise summary of more than a hundred parallelization tools.

Several aspects of the parallelization problem (on distributed-memory machines) are known to be NP-complete. The problem of finding optimal data storage patterns for parallel machines was shown to be a NP-complete problem by Mace [23]. Li and Chen [22] have shown that the related problem of data-alignment (mapping array dimensions to processor dimensions) is also NP-complete.

In the early days of computing on distributed-memory machines, scientists wrote message-passing programs and had to deal with the issues of data-partitioning, data-alignment, communication, and synchronization. This was error-prone and tedious and a host of data-parallel languages were developed to alleviate this problem. These languages usually have special directives for specifying data layout and parallel loops. Programmers specify data-partitioning and expose parallelism using data-parallel language. The compiler then analyzes the program and automatically inserts communication constructs in the code. Research of this type include: Caper, CC++, Charm, Code2.0, Cool, Dino, Force, Fortran 90, Fortran D, Fortran M, Grids, HPF, Hypertool, Jade, Linda, MeldC, Mentat, Modula-2, OOFortran, P-Languages, Parallax, PC++, PDDP, P-D Linda, Sisal, SR, Strand88, Topsys, Vienna Fortran, Visage, and X3H5. References to all these languages can be obtained from Cheng's survey[6].

The SUPERB project [44] at the University of Vienna was the first automatic parallelization system for distributed-memory systems. Several groups have worked since then to provide compilation systems for data-parallel languages. Some of the prominent systems are: Kennedy's Fortran-D [10] at Rice University, Fox's Fortran 90D/HPF compiler[13] at Syracuse University, Banerjee's PARADIGM [28] project at UIUC, Padua's POLARIS [31] project at UIUC, ADAPTOR [1] from GMD Laboratory for parallel computing,

the Vienna Fortran [40] Compilation System, sHPF [33] Compilation System, Halloron's Fx project [14] at CMU, and Portland Group's F90/HPF [30] Compiler.

Several groups are working to develop compilers to translate sequential codes to parallel codes. Banerjee's PARADIGM [28] compiler accepts sequential Fortran-77 and attempts to produce an optimized message-passing parallel program. Lam's SUIF [36] compiler incorporates inter-procedural analysis, pointer analysis, and automatic computation and data partitioning. However, many years of research suggest that full automation of the parallelization process is an intractable problem. Consequently, the emphasis of recent research has been on developing interactive tools requiring assistance from the user. Kennedy's D-Editor[20] and Applied Parallel Research's FORGExplorer[12][16] are examples of state-of-the-art interactive parallelization tools.

However, none of these tools can do automatic parallelization of complex legacy codes such as MM5. Our experiences with these codes suggest two fundamental reasons for the serious difficulties encountered by the existing tools. One reason is that the research on automatic parallelization is predominantly focused on parallelization of arbitrary sequential programs. Several aspects of the parallelization problem are known to be NP-complete [22] and the parallelization problem is too difficult at this level of generality.

Legacy codes are very complex: the physics of the problem, the mathematical model, the numerical techniques, the optimization techniques used by the programmer - all contribute to the complex semantics. The second reason for the difficulties encountered by the existing tools is that they depend mainly on information gathered from syntactic analysis of programs and lack an effective way of dealing with the complex semantics of legacy codes.

We have developed a new approach to automatic parallelization [29] which focuses on specific classes of codes and uses high-level knowledge about these codes to overcome these difficulties. We have come across studies, some related to parallel programming and others related to software engineering environments, which explore the use of high-level knowledge in development of software. An empirical study of types of programming knowledge used by expert programmers was reported in Soloway [38].

In software engineering, the use of high-level knowledge is being investigated for a number of years to improve productivity and quality of software. The CHI project at Kestrel Institute [35] and the Programmers Apprentice project at M.I.T [32] are well known examples

of this type of research. Although these projects are not directly related to our goals, some of the basic concepts such as intelligent assistance, and incremental automation were useful in developing our framework. The Proteus system (Goldberg[18]), uses a knowledge-based program development tool to translate subsets of Proteus language constructs. Starting with an initial high-level specification, Proteus programs are developed through program transformations which incrementally incorporate architectural details of a parallel machine. The Linda Program Builder (Ahmed [2]), supports templates which serve as a blueprint for program construction. The programming environments such as Dime [26], and the mesh computation environment [23] use a parallel program archetype to develop parallel applications with common computation/communication structures by providing methods and code libraries specific to that structure. However, the systems and programming environments that we have come across do not provide a framework for parallelization of existing code.

CHAPTER 3 NEW APPROACH

3.1 Introduction

Existing parallelization tools fail to work on legacy codes. We have developed a new approach to automatic parallelization [29] which overcomes the difficulties faced by these tools. This approach has three main components: the concept of class-based parallelization, a technique for expressing and analyzing parallel mappings, and a structured parallelization process. These components are described in the following sections.

We use the SPMD (Single Program Multiple Data) model of parallel computation, whose characteristics can be captured by the BSP [39] model or a special case of Synder's [37] model. Processors proceed together through a sequence of steps; although within a step different processors may take different execution paths. Each step is followed by a sync/ exchange point. A request to fetch or store a remote data item can occur anywhere within a step but it is not guaranteed to be satisfied until the end of the step, when all processors are synchronized.

3.2 Class-Based Parallelization

3.2.1 Fundamental Idea

The new approach has one fundamental dogma: focus on codes that belong to a class. For example, focus on codes that are based on one of the following methods: finite-difference (FD) method, finite-element method (FEM), and boundary-element method (BEM). All codes based on a particular numerical method are said to belong to the same class. This approach has wide applicability because the vast majority of scientific and engineering codes are based on one of the above methods. Note that this approach is a departure from existing tools which attempt to parallelize any arbitrary sequential code.

The advantage of focusing on a class of codes is that the class-specific high-level knowledge can be used to simplify the otherwise intractable problem of parallelization. For each class of codes, the analyses and transformations to arrive at the parallel code are

tailored to make use of the characteristics of the numerical method corresponding to the class. This is explained in section 4 and again in the next chapter.

During conversion of a typical legacy code to a parallel code, most of the serial code can be re-used as is; In other words, only a few portions of the serial code need to be changed to convert it into a parallel code. High-level knowledge about the class helps to identify and focus on these critical portions of the legacy code and thus provides an effective way to deal with its complex semantics.

3.2.2 Abstract Representation Of A Class

An *abstract representation* of a class attempts to capture those computational characteristics that play an important role in designing a parallel algorithm. Consider matrix multiplication: C = A X B. The abstract representation is shown below:

Atomic computations: comp (i,j,k)

Computation Set: S = {comp(i,j,k) | 0 <= i,j,k < N}

Variable access pattern: Read Set = {(i,k), (k,j)} and Write Set = {(i,j)}

The variable access pattern, specified by the Read Set and the Write Set, shows how the read and write variables are accessed by the computation. Note that, unlike a for loop construct, the abstract representation does not prescribe any artificial fixed order for performing the computations.

The abstract representation is useful for focusing on what is the computation as opposed to how it is to be done. A sequential program is a homogenized combination of what is to be done and how it is to be done [6]. The sequential expression of the how part introduces artificial dependencies [34] and often reflects the underlying compiler technology and machine structure. The how part is irrelevant and distracting for uncovering the parallelism. For the purpose of parallelization it is important to focus on the what part. The separation of what from how is usually very difficult without using high-level knowledge of the computation. The abstract representation embodies the high-level knowledge necessary for disentanglement of sequential and parallel by identifying atomic sections of the code that can be embedded in a parallel program as the code to be executed sequentially at each individual processor. As explained earlier, it is possible to reuse large sections of the sequential code in the parallel code. For example, the sections of code associated with physics calculations within an individual grid cell can often be directly embedded into a parallel program.

Another use of the abstract representation is to simplify dependency analysis (analysis to find out the ordering of computations and also which computations are independent and hence can be done in parallel). Identification of sections of the original sequential code as atomic segments hides numerous basic blocks inside a single atomic code segment (explained later in chapter 4), and thus simplifies the dependency analysis. In contrast, the commonly followed approach to dependency analysis based on basic blocks defined by the control flow of the program, when applied to large production codes, results in a very large number of basic blocks and creates an explosion of information for analyzing dependencies.

As explained in the next section, another use of the abstract representation is to help to develop parallel mappings.

3.3 Parallel Mapping

3.3.1 Functional Representation Of A Parallel Mapping

A paradigm for representing and analyzing parallel mappings was developed in [21]. The representation is in terms of functions that maps the *computational space* of the abstract representation of a class to the space formed by cross product of *processor space* and the *time dimension* for representing parallel time steps. Unlike this representation, a typical approach to data parallelism does not explicitly capture the behavior of the parallel mapping along the time dimension. For example in High-Performance FORTRAN (HPF), the data decomposition is specified and the owner-computes rule is used. Thus the specification deals with only distribution of computations among processors.

In practice, the number of processors is typically much smaller than the number of computations to be done at each parallel step. To handle this situation, computations are grouped together and assigned to a single processor. One way to specify the grouping, as done in HPF, is to use data decomposition schemes such as block or scatter decomposition. An equivalent way to view the grouping of computations is the concept of processor virtualization where virtual processors are grouped together to correspond to one physical processor. The grouping scheme can then be represented by a function that maps virtual processors to physical processors.

The representation of the grouping scheme along with the original representation of the parallel mapping abstractly defines the execution profile at each processor. By including the *time dimension* in the representation one can capture the dynamics of a parallel mapping. Based on the dynamics, the trajectory of an array variable can be determined and hence it's communication pattern can be deduced. By using the grouping information, it is possible to develop an analytical method for describing the computational balance across the processors.

Usually, the number of valid parallel mappings (mappings leading to correct parallel algorithms) is very large which makes the search for efficient parallel algorithms a difficult problem. For instance, it can be shown that the number of valid mappings for matrix multiplication on a two dimensional mesh of processors is bigger than $(n!)^P$ where P is the number of processors and the matrix size is n X n.

3.3.2 Search For Efficient Parallel Mapping

We are developing an automatic method for arriving at functional representations corresponding to efficient parallel mappings which is based on linear algebra and diophantine analysis. There are two important special characteristics, covering many numerical algorithms, which can be very effectively exploited by a mathematical method. The first is that the array variable indexes and the do loop indexes are affine functions of basic parameters defining the computational space. The second key characteristic is that the computational space can be represented by a convex set and in most cases the convex set has a very simple structure such as a regular polyhedron.

In order to minimize the parallel execution time, load balance and communication are the main issues. The high-level parallel mapping is defined with respect to a logical array of arbitrarily large number of processors. The parallelization process proceeds with the logical array until it reaches the point where it must decide how to partition the data arrays among a finite set of processors. To partition the data arrays, first the computations are grouped together and each group is assigned to one processor. The data partitioning is chosen so that it is consistent with the scheme for grouping computations. The grouping scheme, thought of as processor virtualization, determines the load-balance. The user can select from common grouping schemes such as scatter decomposition and block decomposition to alleviate load imbalance.

The search for efficient parallel mappings can be effectively constricted by using our approach. The functional representation of a parallel mapping, by capturing the dynamics of the mapping, can determine the trajectories of array variables accessed in the computation. The ability to determine the trajectory can be used to constrict the space of parallel mappings in a way that leads to efficient parallel mappings. For example, one may allow only those parallel mappings whose functional representations lead to linear trajectories. Such trajectories will typically enforce nearest neighbor communication which is a desirable property for generating efficient parallel algorithms. Another mechanism for generating efficient parallel mappings is by minimizing the span along the time dimension while keeping a fixed bound on the number of processors. Intuitively, the span is the number of steps in the parallel algorithm and the length of the critical path is the lower bound for the span.

In some cases, finding an efficient parallel mapping can be automated. However, the need for an automatic method to generate an efficient parallel mapping depends on the problem. There are cases where it may be best to let the user specify the parallel mapping. The user may have high-level knowledge of the problem domain to suggest an efficient parallel mapping and that knowledge may be difficult to replicate in an automated system. For example, in atmospheric models such as MM5 the physics calculations generate maximum data exchanges in the vertical direction. Based on this knowledge, the user can specify the parallel mapping to be the projection of the 3-D computation grid along the vertical direction onto a 2-D processor array. This choice of the projection direction will lower the inter-processor communication.

3.4 Structured Parallelization Process

A structured process is necessary to manage the complexity of the parallelization task. Figure 1 shows the structured parallelization process used by our approach. The system requires the user to specify the high-level knowledge but it automates tasks which are time-consuming, tedious, and error-prone. The process blends automation and user assistance to provide a pragmatic solution for parallelization of specific classes of codes.

The starting point for the parallelization process is to have the user specify the class to which the code belongs. Our approach follows the strategy used by experienced parallel programmers. The programmer first understands the algorithmic form and then uses that

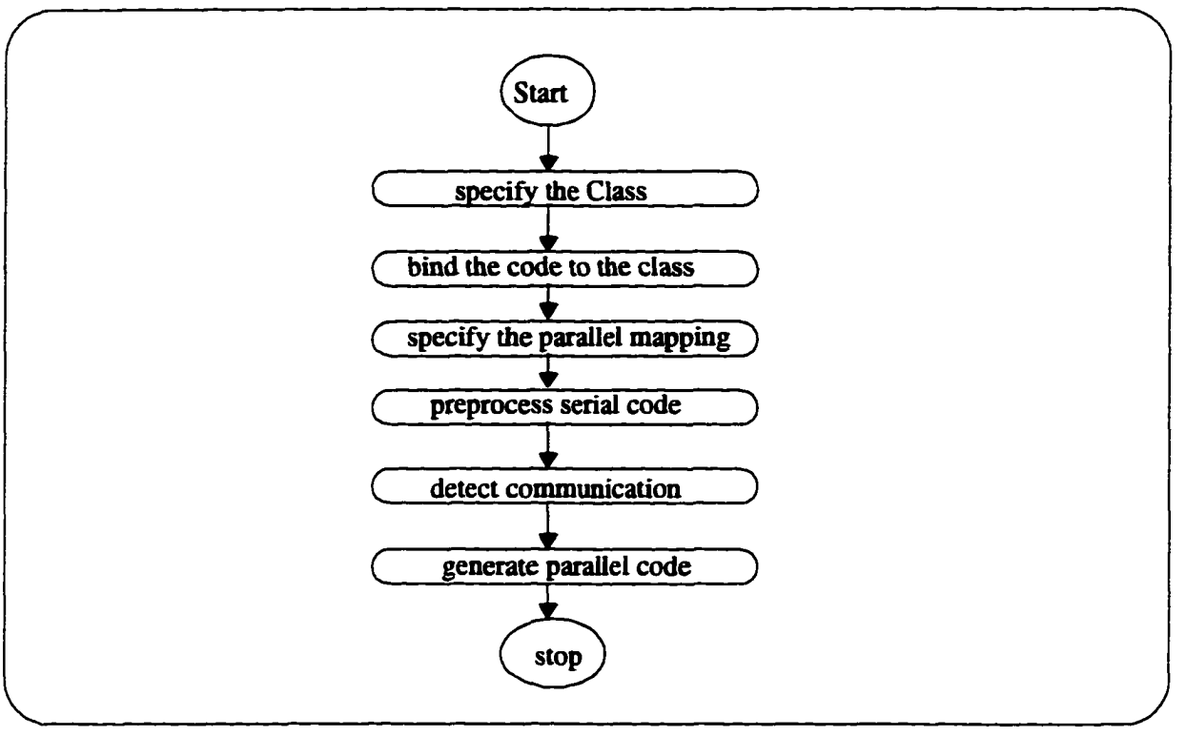


Figure 1: Structured Parallelization Process

knowledge to arrive at an efficient parallel code. For instance, the programmer will first find out that it is a finite-difference code and then proceed with the parallelization. This crucial step presents a type of pattern-recognition problem which is beyond what can be automated with existing technology. The user, however, can readily assist a parallelization tool by identifying the form of the algorithm.

The next step is to provide a binding between the code and the class by specifying the spatial and temporal indexes used in the code. This binding helps the system to identify and focus on the parts of the serial code that are relevant for the purposes of parallelization. This binding is explained later in subsection 4.1.2.

The third step is to determine the parallel mapping. For complex problems such as MM5, the user specifies the parallel mapping and the system validates the mapping through dependency analysis. The alternative used in other approaches is to use the dependency analysis to search for a parallel mapping. Thus, in both cases dependency analysis is required but the goals are different. In this case, the validation is much simpler than the

search. Prime factorization is a good example to understand the difference in the level of difficulty. Validating a parallel mapping is like checking if a given factorization of a number is correct and searching for a parallel mapping is like finding the prime factors. The latter problem is much harder.

The fourth step is to preprocess the sequential code. This is quite different from what is done in a typical compiler framework. A major difference is that the preprocessing includes steps to check if the given code conforms to the abstract representation of the class identified by the user. The system can automatically determine quirks in the code that can create problems for efficient parallelization. These quirks often reflect ad hoc programming practices adopted in the serial code to suit specific compiler technology or machine architecture.

The fifth step is to determine the communication requirements. The communication cost has three components: contention, latency, and volume. Contention depends on the data access patterns; these are determined by the parallel mapping and the initial data layout. Latency and volume components are reduced by collapsing the exchange points. Typically, the latency is high because of the high-message start-up cost. Therefore, there is an advantage in collapsing the number of exchange points and aggregating smaller messages into a single large message. A code may have hundreds of exchange points. In practice, these are collapsed into a small number of exchange points. The communication is specified at a logical level and the low-level details of communication such as packaging of messages is handled by a run-time library.

The last step is to generate the parallel code. Modifications to serial code include: insertion of data decomposition primitives, transformation from global to local loop indices, and insertion of communication primitives. The end result is SPMD code for a distributed memory machine using mesh topology for communication.

Details of the structured parallelization process and its implementation for the class of explicit FD codes is presented in the next chapter.

CHAPTER 4 PARAGENT

parAgent (Parallelization Agent) is a parallelization tool, built using the new approach described in the previous chapter. This tool converts serial legacy Fortran-77 codes that are based on the regular-grid-based explicit time-marching FD model to SPMD parallel code. In the next section, we describe the details of the parAgent software modules.A later section describes the parallelization process used by parAgent.

4.1 Description Of parAgent

parAgent is comprised of seven modules: Parser, Array-space map Builder, Conformity Checker, Block Generator, Communication Analyzer, Inter-procedural Handler, and Parallel code Generator. Each module (Figure 2) is described in the subsections below. A Graphics User Interface (GUI) provides a user-friendly interface to parAgent. The various screens of the GUI guides the user through the parallelization process, reports errors and inconsistencies, and provides crucial information about the code such as the call-tree and the blocks and stencils display.

4.1.1 Parser

The current parser is capable of processing Fortran-77 files to obtain the information that is needed by the other modules. Four basic types of information are obtained from the code: variable identification information, read-write information, control-flow information, and caller-callee information.

The ***variable identification*** information consists of:

1. Names of all variables, their scope (i.e. whether local, global, or function-argument), their type (i.e. whether a scalar or array variable), and the number of dimension for each array variable.
2. For each array variable, subscript information for each use of the array variable in the entire code.
3. Names of procedures and their formal parameters.

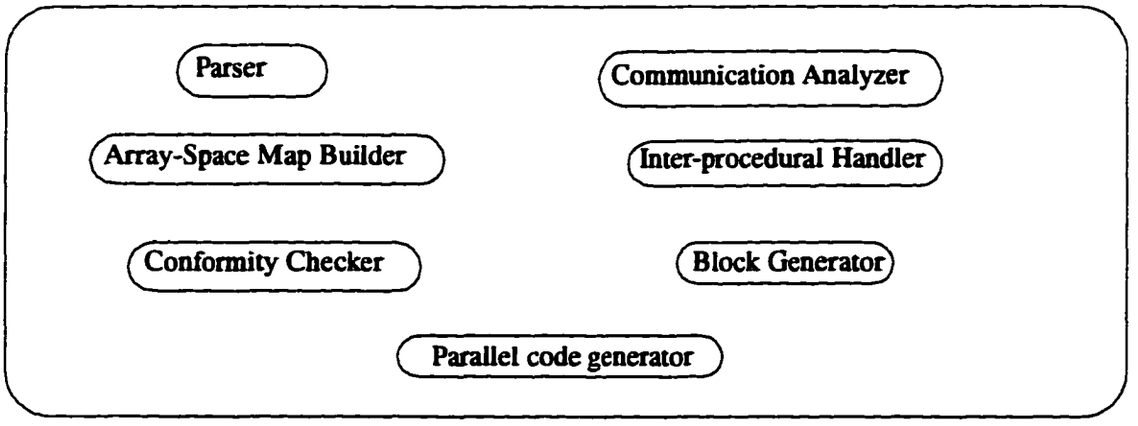


Figure 2: parAgent

For each line of code, the ***read-write*** information consists of the names of variables that are read from, the names of variables that are written to, and subscript information for each use of array variables in the line. The ***control-flow*** information identifies the start and end of if-then-else statements, do statements, go-to statements, and subroutine calls. For each procedure call, the ***caller-callee*** information consists of the name of the procedure being called, and the actual arguments being passed to the procedure. For each procedure (the caller), the names of all the procedures (callees) that are called are stored.

4.1.2 Array-space Map Builder

The array-space map builder provides a binding between the code and the class by specifying the spatial and temporal indexes used in the code. In sequential codes based on regular-grid-based FD model, each dimension of the grid is represented by some loop index, such as i, j, or k (say), as shown in the Figure 3. A reference to the array variable B[k][j][i] implies that the grid dimension represented by loop index k corresponds to dimension 1 of the array B, the grid dimension

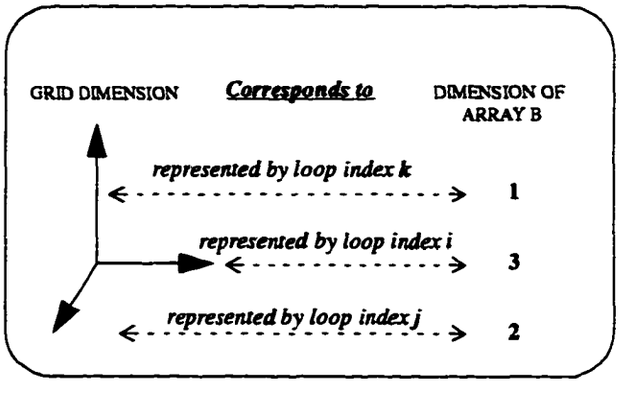


Figure 3: Array-Space Map

represented by loop index j corresponds to dimension 2 of the array B, and the grid dimension represented by loop index i corresponds to dimension 3 of the array B. It is expected that this correspondence between the grid dimension and the array dimension for the Array B remains the same throughout the code.

The Array-space map builder collects information about the correspondence between the grid dimension and array dimension. Each array use is scanned for occurrence of the loop index i, j, or k in each of the subscripts. If an index is found in an array dimension, then the grid dimension represented by the index is said to correspond to the array dimension. This information is stored for future reference (see Table 1).

Table 1: Array-space Map example

Array use in code	grid dimension i corresponds to array dimension	grid dimension j corresponds to array dimension	grid dimension k corresponds to array dimension
A[i+1][j-2][32]	1	2	???
B[j+3][43][k-2]	???	1	3
C[v][m][i+2]	3	???	???

From the table, the grid dimension represented by i appears to correspond to array dimension 1. A later use of the array A, say A[1][i+2][32], could reveal that loop index i (i.e. the grid dimension that it represents) also appears to correspond to array dimension 2, or that loop index j and loop index i both correspond to array dimension 2. This inconsistency implies that at different sections of the code, the loop index i has been used to represent different grid dimensions.

The Array-space map builder catches conflicts of this nature and generates a list of such conflicts. User interaction is required to resolve these conflicts. In the above example, the user could change the code so that loop index i represents always the same grid dimension throughout the code.

Sometimes the Array-space map builder is unable to find any correspondence between a grid dimension and an array dimension (denoted by??? in Table 1) for an array A (say). User interaction is required at this point to either verify that no such correspondence

exists for the array A, or to specify it manually. The array-space map information is stored in a file and this information needs to be found only once for each variable.

4.1.3 Conformity-Checker

All codes for explicit FD models have certain characteristics (called Class characteristics). Also, to simplify the analysis of legacy codes, parAgent assumes that these codes adhere to some rules (called Standardization rules). The conformity-checker verifies that the serial code coheres to the Class Characteristics and the Standardization rules. It flags each violation and mostly lets the user change the code so as to make it conform (See Figure 4). It is often the case, that it is easy for the user to make a few changes and make the code conform - than to have parAgent try to figure out a fix for the problem. A complete list of FD Class characteristics and the Standardization rules is provided in the Appendix.

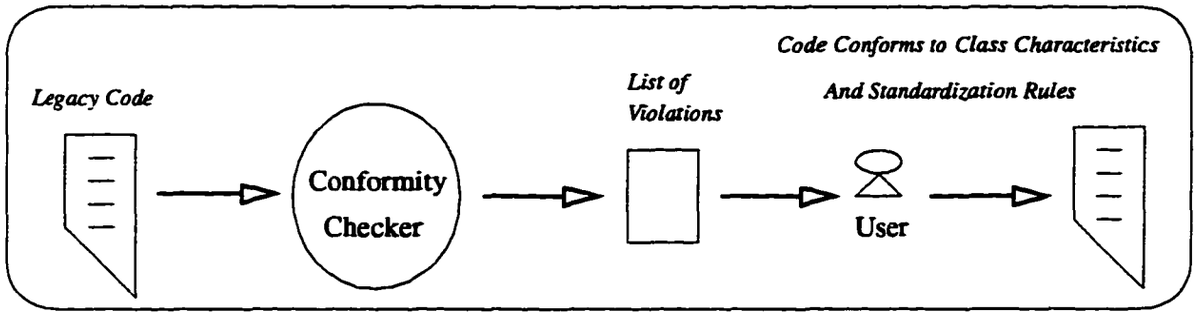


Figure 4: Conformity Checker

CLASS CHARACTERISTICS. One of the Class Characteristics for codes based on explicit FD model is that these codes have near-neighbor communication pattern. Consider the statement b = A[2,3]. The scalar b is assumed to be replicated on all nodes. Assume that the grid dimension represented by loop index i, corresponds to Array A's dimension 1, and that the grid dimension represented by loop index j, corresponds to Array A's dimension 2. Then A[2,3] would lie on node (2,3). The contents of A[2,3] would be needed at all the places that b resides - i.e. at all the nodes. Thus the statement b=A[2,3] implies a broadcast communication. The conformity-checker would flag this statement as it violates the near-neighbor-communication requirement.

In one scenario, the programmer may have associated the scalar variable b with the grid point (2,3) and always used b in that context. In this case, the user could introduce an

array B and replace the statement b = A[2,3] by the statement B[2][3] = A[2][3]. Also, all instances of b would need to be replaced by B[2][3]. After this change, the statement would no longer imply a broadcast to parAgent and would pass verification. Note that it would be hard for parAgent to figure out the fix for this situation.

STANDARDIZATION RULES. Legacy code tend to be abstruse. There are several reasons for this. One reason is that the programmer makes changes to the code to get the best performance out of the underlying machine architecture. For example, to get the best use of cache memory, the programmer may have used blocked algorithms. Another reason is that these codes have evolved over a long period of time. During this time, multiple programmers, with different coding styles and varying levels of expertise, make changes to the code, with an intent to fix bugs, add functionality, and make improvements.

To simplify the analysis of legacy codes, parAgent assumes that these codes adhere to some rules (called Standardization rules). For example, a central assumption is that the communication is statically determinable. Another assumption is that arrays are not aliased to lower dimensions. For example, a 2D array cannot be used like a 1D array - although that would be valid Fortran code. Some of the Standardization rules could be relaxed in enhanced versions of parAgent.

4.1.4 Block Generator

The block generator performs the following tasks:

1. converts the code to an internal block representation
2. computes the read/write characteristics of the block
3. performs conformity checks at the blocks level

Many thousands of lines of code can be usually represented by a few blocks. This has several advantages. First, further analyses operate at the block level - rather than at the statement level - and this reduces the computational complexity. Second, the blocks representation (as displayed by GUI) shows the communication points in the code. Note that the cost of communication of data between processors depends on the number of such points. Third, the user can understand the underlying structure of the serial code as pertaining to parallelization and can make educated choices to obtain efficient parallel code.

For example, the user could decide to relocate some code segment to reduce the amount of communication.

There are two types of blocks in this internal representation: Container blocks and Atomic blocks. An Atomic block consists of one or more lines of code. Container blocks consist of Atomic blocks and "child" Container blocks.

An Atomic block or a Container block is not the same as a Basic block normally encountered in a typical compiler context. Control flow statements exclusively determine creation of Basic blocks, whereas this is not true for Atomic or Container blocks. Atomic blocks normally contain several lines of code, including multiple control flow statements (see Figure 5).

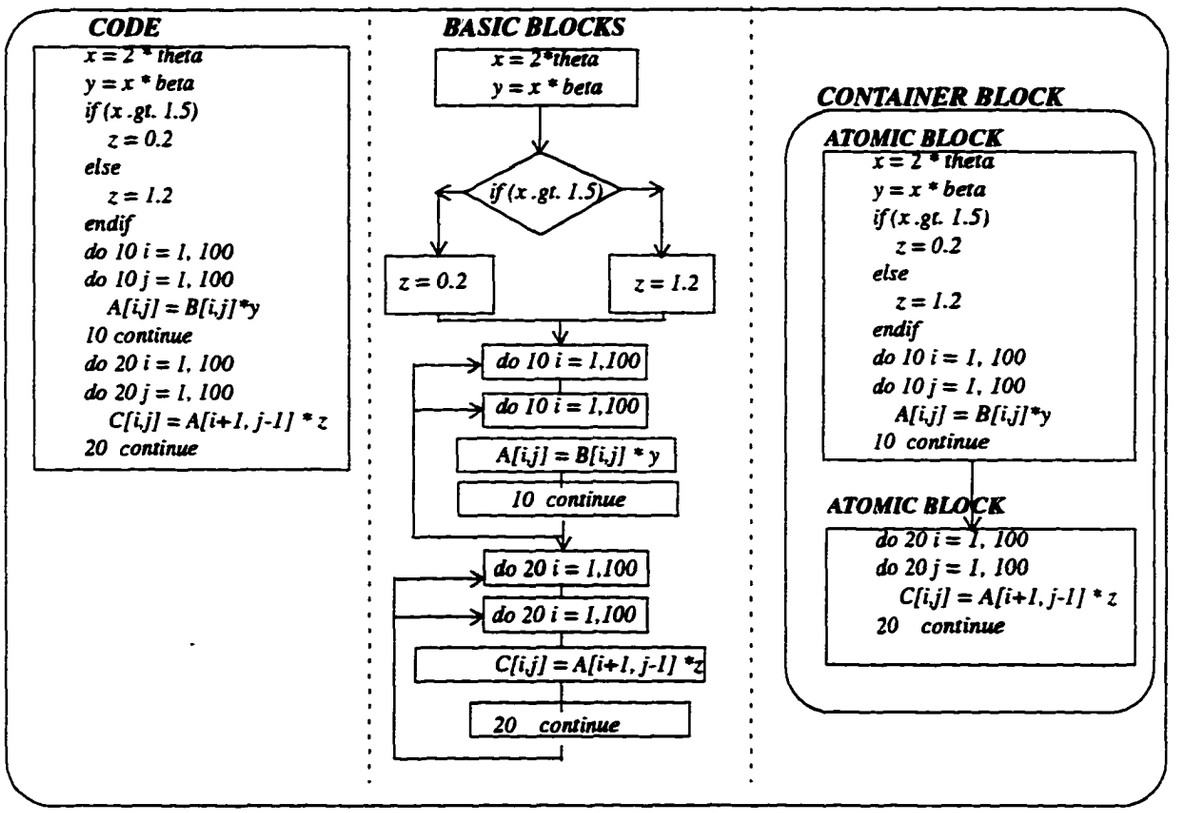


Figure 5: Atomic Blocks

The blocks generator goes over the code forming Container and Atomic blocks. Whenever possible, blocks are combined together to form larger blocks. When forming blocks, the block generator computes the overall read/write characteristics of each block from

the read/write characteristics of all the lines that make up the block. The rules for forming and combining blocks are described below.

Combining Blocks to Form larger blocks:

1. A Container block which has only one Atomic block is converted to a larger Atomic block.
2. Two adjacent Atomic blocks, such that there is no write variable in the read/write characteristics of the upper Atomic block that corresponds to a read variable in the read/write characteristics of the lower Atomic block, are collapsed to form a larger Atomic block.
3. An if-Container block has a then-Container block and an else-Container block enclosed in it. These are all collapsed to form a larger Atomic block only if the then-Container block and the else-Container block have only one Atomic block each and if these Atomic blocks have the same write characteristics.
4. When two blocks are combined to form a larger block, the read/write characteristics of the blocks are also combined to form the read/write characteristics of the larger block.

Container Blocks:

1. A Container block is started when a control-flow statement (i.e. subroutine, if, then, else, and do) is encountered.
2. The Container block is ended when the control-flow statement ends (i.e. end, endif, enddo).

Atomic Blocks:

1. An Atomic block is started at the first non-control-flow executable statement that follows the start of a Container block. At this point, the Atomic block has a read/write characteristic which is identical to that of the executable statement.
2. The next executable statement is added to the Atomic block if both the following conditions are true: a) it is a non-control-flow statement, and b) there is no write variable in the read/write characteristics of the Atomic block that corresponds to a read variable in the read/write characteristic of the statement. Note that Call statements are treated like a non-control-flow statement. The read/write characteristics

of the call statement is determined from a previous pass of the callee by the parAgent.

3. If the next statement does not satisfy the conditions stated above, then the Atomic block is ended.
4. When a statement is added to the Atomic block, the read/write characteristics of the statement is added to that of the block to form the blocks new read/write characteristics.

Note that there are some constraints that must be satisfied by the blocks. The conformity-checker, which operates on a line-by-line basis, is unable to perform these checks. Thus, the blocks generator also performs conformity checks on the blocks. For example, DO container blocks which are indexed by i or j (chosen to be directions of parallelization) can have only one atomic block in them. If two Atomic blocks are found in such a DO container block, parAgent handles the situation by first alerting the user to the situation - and then by splitting the container block into two container blocks if permitted to do so by the user. This situation corresponds to a typical loop-split encountered in typical parallelizing compilers.

4.1.5 Communication-Analyzer

For each subroutine, the communication analyzer determines the read/write information of the entire subroutine, and also information about any communication that may occur within the subroutine.

The read information is expressed in terms of the subroutines formal parameters and global variables, corresponds to the array variables that need to be read in prior to the start of the subroutine. The write information, also expressed in terms of the subroutines formal parameters and global variables, corresponds to the array variables that were modified within the subroutine. The read/write information is used by the caller, which substitutes the formal parameters by the actual arguments used to make the call. The call is then treated just like any other non-control-flow executable statement by the block generator.

Information about communication that may occur within the subroutine is given by which array-variables need to be transferred, from which processor, to which processor, and at what point in the code. The communication overhead is a function of the number of times communication takes place and how much data is sent. Coalescing messages to the same

processor is a standard way to reduce the communication. It is possible for the user to spot further opportunities for reducing communication by performing code relocation.

The analyzer uses the block representation of the code to gather all this information. The block representation is as if the code has been converted to a very high level language. The block representation reduces the computational complexity. Each atomic block is like a line of code with its read and write information. The container blocks contain the control-flow information. The analyzer uses standard compiler def-use and use-def analysis to determine the communication requirements of the code.

An underlying assumption in using the block representation, is that the program has structured code. However, this assumption is not binding and it is possible to use an iterative technique to determine the communication requirements of the code. This is not implemented in the current version of parAgent.

4.1.6 Inter-procedural Analysis Driver

The Inter-procedural analysis driver performs two tasks:

1. forms a call-list from a post-order scan of the call tree formed by the parser, and
2. drives analysis of each subroutine as they occur in the call-list.

Each subroutine is passed through the parser, array-space map builder, conformity-checker, block generator, and communication analyzer. The read/write information of the entire subroutine is computed. Also, information about any communication that may occur within the subroutine is obtained. The subroutines are processed in call-list order. This ensures that when a subroutine is being processed, all subroutines that are called from it have already been processed.

As an example, see Figure 6. In the figure, as per the call-list, subroutine D is processed first and then subroutine E and so on. When subroutine B is being processedthe two subroutines that are called from B (i.e. D and E) have already been processed, and are replaced in B by their read/write characteristics.,

4.1.7 Parallel Code Generator

After analysis of the code, parAgent provides the option to generate the parallel code. This involves making changes to the serial code so that the code would actually run on a parallel computer.

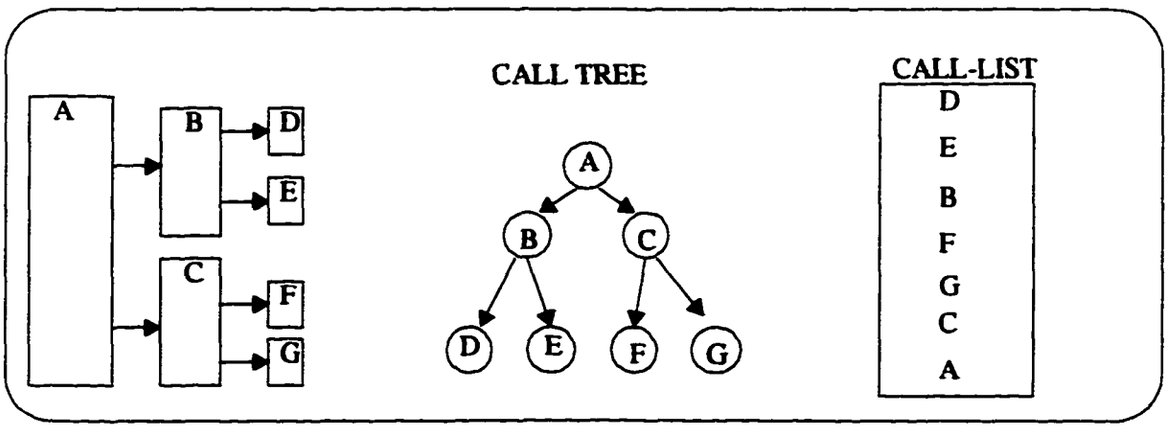


Figure 6: Call Tree

The current version of parAgent makes use of the Runtime System Library (RSL) [26] (developed at Argonne National Laboratory). RSL provides an abstract and high-level interface to the parallel machine, simplifying the programming task. Buffer allocation, copying, routing, and other details of the underlying message-passing mechanism are encapsulated within high-level routines for stencil exchange. Note that future versions of parAgent could use some other communication library (for example: MPI) for generation of parallel code.

Generation of parallel code consists of: creation of a driver routine, creation of communication stencil definitions, and modifications to the serial code. The driver routine (See Figure 21) invokes RSL initialization routines, and the parallel code. The driver routine also allows the user to configure the parallel program according to the computer resources available to the user. In particular, the processor array size can be configured. Using RSL, communication is specified abstractly as stencils, which RSL converts to the appropriate low-level communications between processors. Communication stencils (See Figure 22 and Figure 23) are declared using RSL calls provided for that purpose.

Some of the modifications that are made to the serial code: arrays are partitioned among processors and so the local sizes of the arrays need to be adjusted, iteration over i and j loop indices are replaced by RSL loop constructs, and calls to communication routines using RSL stencils are inserted at appropriate places (as determined by prior analysis by parAgent). Changes occur mainly to control-flow statements involving either i or j indices. All other control-flow statements and all computational statements remain unchanged. The example in Chapter 5 Section 7 illustrates parallel code generation.

4.2 The Parallelization Process

This section describes the steps that a user would take to parallelize a serial code using parAgent. There are two main phases. In the first phase, the diagnostic phase, parAgent extracts the array-space map information and resolves conformity violations with the help of the user. The second phase, the parallelization phase, is mostly automatic and proceeds with analysis of each subroutine - forming atomic blocks and finding communication information. After this phase, the user can invoke the code generator to generate parallel code.

4.2.1 Diagnostic Phase

STEP 1: The user first chooses the FD class. He is presented with the *enquiry screen* (Figure 7) and indicates the location of the source files and the binding of the code with the class. This step corresponds with the steps "specifying the class", "binding the code", and "specifying the parallel mapping" from the structured parallelization process described in chapter 3.

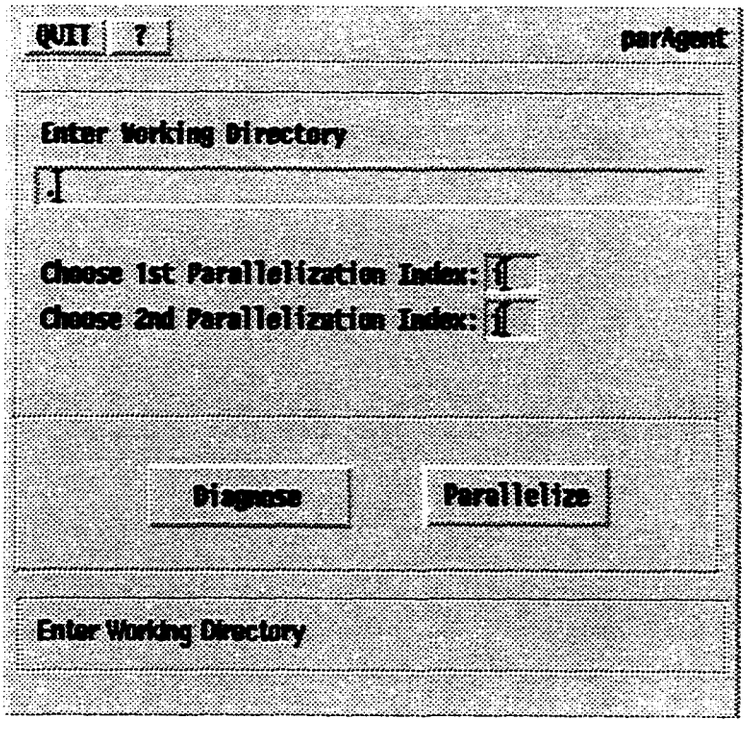


Figure 7: Enquiry Screen

STEP 2: The user is presented the *diagnosis screen* (Figure 8). After selection of a file, he selects the *diagnose* button to run diagnostic checks on the file.

STEP 3: parAgent starts to collect information for each array - about the correspondence between grid dimension and array dimension. This is explained in the section on the description of parAgent. Unresolved correspondences and conflicts are resolved by the user. The user is presented with a list of unresolved problems. After the user has made the necessary changes, he has to repeat this step until there are no unresolved problems.

STEP 4: At this point the *conformity-checker* verifies that the code coheres to the Class Characteristics and Standardization rules. Any violation is detection and presented to the

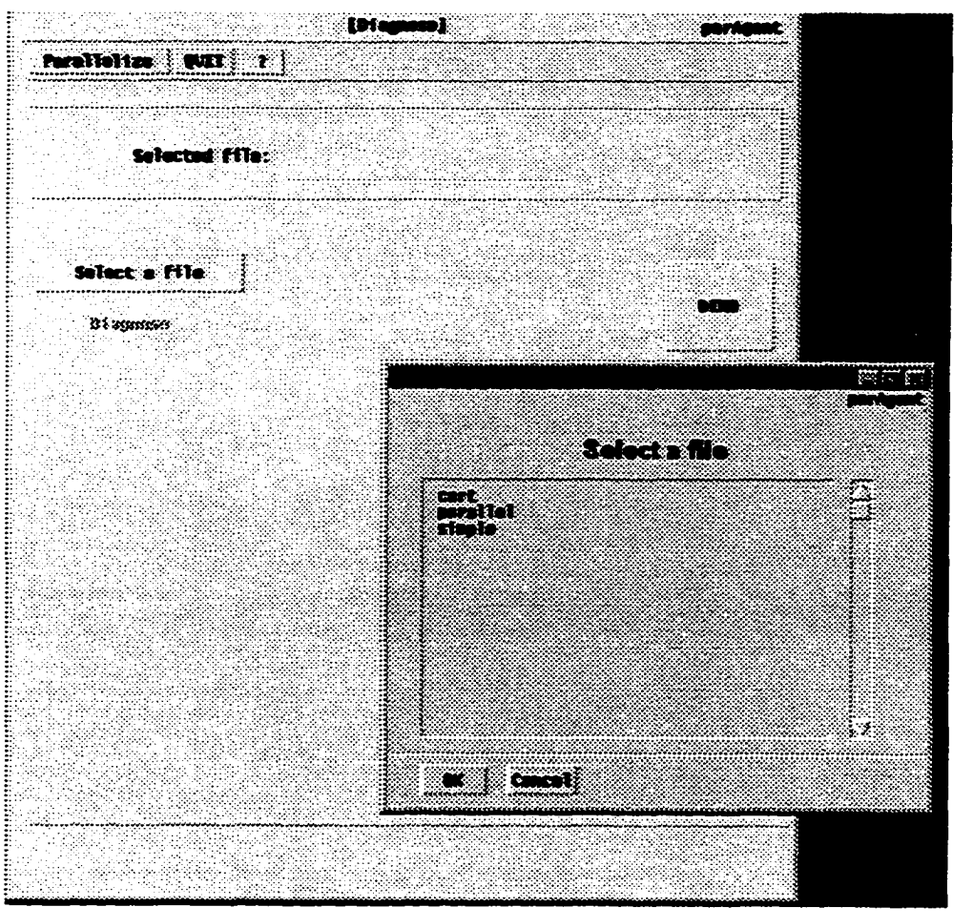


Figure 8: Diagnosis Screen

user in form of a list. After the user has made the necessary changes, he has to repeat his step until there are no violations.

STEP 5: The user proceeds with steps 1 through step 4 for all source files. The results of the diagnostic steps are saved in the source directory.

4.2.2 Parallelization Phase

STEP 1: After diagnosis is completed, the user is presented with the *parallelization screen* (Figure 9). There are two modes of parallelization. In the first mode, inter-procedural analysis takes place and all subroutines get parallelized. In the second mode, only the selected file is parallelized. In this mode, all subroutine calls in the function are ignored. The user can choose one of the modes from the parallelization screen.

STEP 2: The user can choose to view the call-tree originating from the selected file. This is shown in a display screen.

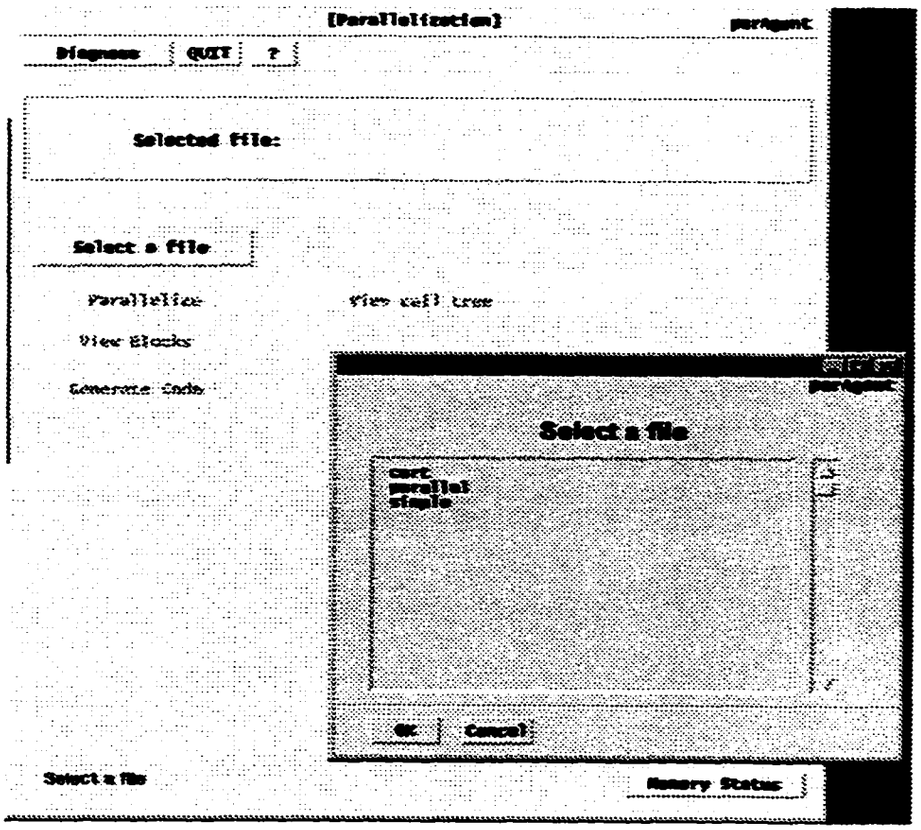


Figure 9: Parallelization Screen

STEP 3: In this step, the *inter-procedural analysis driver* guides the parallelization, of each subroutine in the source file, in call-list order. Each subroutine is processed by the *block generator* and then by the *communication analyzer.*

STEP 4: The user can now choose to view the blocks and communication points, and the communication stencils found from analysis of the code.

STEP 5: At this step, the user can choose to generate parallel code for the analyzed files. The *parallel-code generator* is invoked to produce the parallel code.

CHAPTER 5 CAPABILITIES OF PARAGENT

The capabilities of the current parAgent are discussed in this chapter. We will present several examples each intended to demonstrate a specific capability of parAgent. The following sections describe each of the following capabilities:

1. Automatic generation of Array-space map information
2. Conformity Checking
3. Analyzing Communication requirements and Display of stencils
4. Communication optimization
5. Inter-procedural analysis
6. Loop Split - detection and action
7. Parallel Code Generation

5.1 Automatic Generation Of Array-space Map Information

parAgent automatically finds the correspondence (if any) between array dimension and grid dimension for each array variable after program loop indices have been bound to grid dimensions. This information is then stored in a ".f.var" file. The examples in Figure 10 and Figure 11 serve to highlight the issues.

Refer to Figure 10. Consider the array variable *a*. It is referred in lines L6, L7, and L12. From these references, parAgent automatically deduces that the grid dimension corresponding to the loop index i maps to array dimension 1 and that the grid dimension corresponding to the loop index j maps to array dimension 2. The array variable *c* is also mapped similarly.

Variable *b* needs special handling. It is referred in lines L6 and L7. From the information in the lines, parAgent is able to deduce that the grid dimension corresponding to the loop index i maps to array dimension 1. However, it is not possible for parAgent to obtain any information about mapping of loop index j. For example, it could be that b has no array dimension which corresponds to the j grid dimension. In this situation, parAgent displays the message shown in the figure and leaves the decision to the user. The message provides the

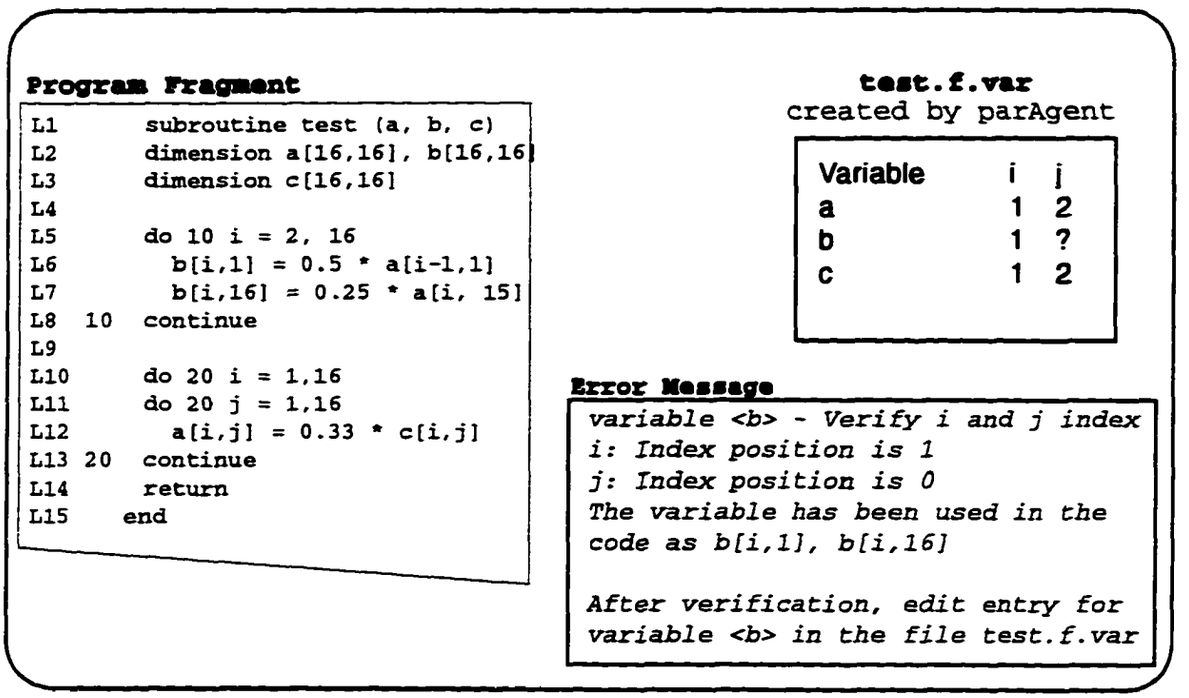


Figure 10: Array-Space Map

information to assist the user to make the decision. The user may then edit the map-index file "test.f.var" and map index j to array dimension 2.

Now refer to Figure 11. The variable c is referred in lines L6 and L10. From these reference in line L6, parAgent deduces that loop index i maps to array dimension 1 and loop index j maps to array dimension 2. On the other hand, the reference in line L10 implies that loop index i maps to array dimension 2 and loop index j maps to array dimension 1. Instead of a one-to-one correspondence between the grid and array dimensions, parAgent finds a many-to-many correspondence. In this situation, parAgent exits with an error message and the user has to change the serial code so that the conflict is removed.

Although the serial code is correct, it does not follow the programming conventions expected by parAgent. The programmer has mapped grid dimension to different loop indices at different parts of the program. In this particular case, the error may be fixed by changing line L10 to

```
L1  20                c[i, j] = 0.33.
```

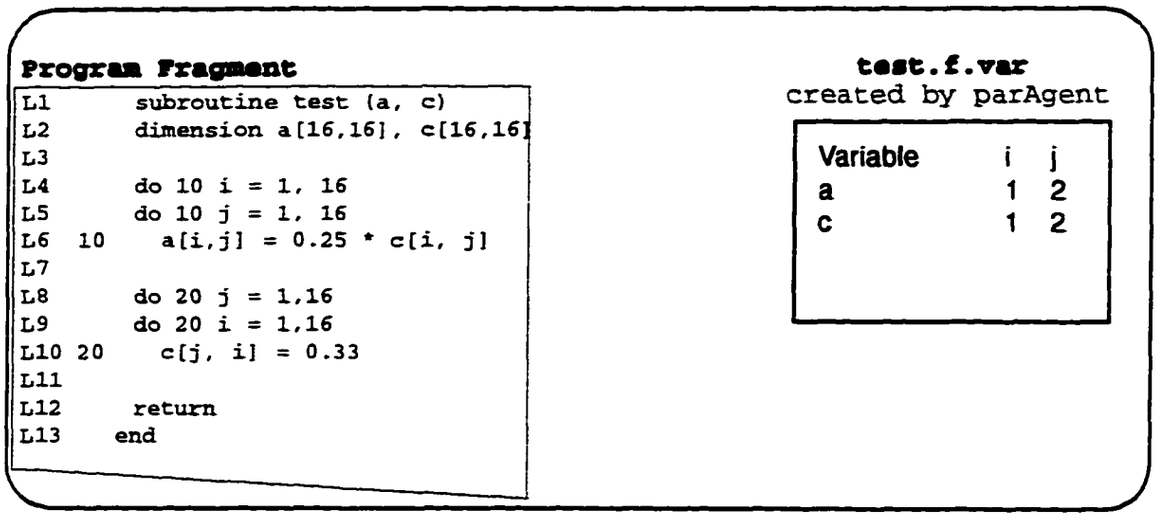


Figure 11: Array-Space Map

5.2 Conformity Checking

parAgent verifies that the serial code conforms to the Class characteristics and Standardization rules (see section 4.1.3). The test code shown in Figure 12 violates several of these rules. For arrays a and c, loop index i is mapped to dimension 1 and loop index j is mapped to dimension 2. For the array b, loop index i is mapped to dimension 1 and that loop index j is not mapped to any dimension. Assume that an array element X[i,j] is stored on processor (i,j), where X is an array variable.

One of the standardization rules is that if an array dimension which is mapped to one of the loop indices i or j, then only expressions of the form i ± c or j ± c, where c is a constant, can be used in that dimension. Consider line L8. Both arrays a and c have a 1 in the dimension mapped to the j loop index. This violates the rule. However, this is not a major problem and parAgent is able to handle this situation automatically. Line L8 is replaced automatically by the lines L8 and L9 in the post diagnostics code.

Now consider the fragment L10 to L12. parAgent follows the owner-computes rule and processor (i,j) performs the computation in line L12. The array b is replicated along processors along the j dimension and partitioned along processors along the i dimension. The computation at processor (i,j) requires a[i,1,k]. Thus, for each iteration of the k loop, processors (i,1) will have to broadcast a[i,1,k] along the j dimension. This violates the near-

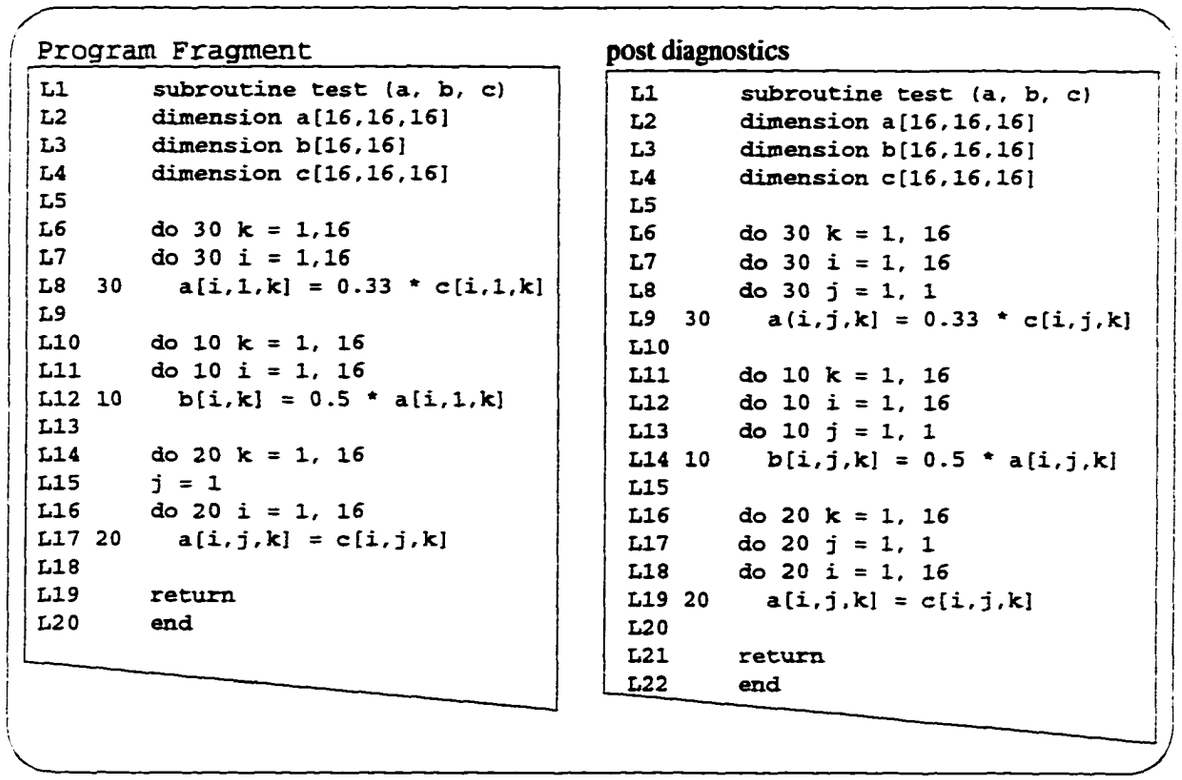


Figure 12: Conformity Check

neighbor communication rule for explicit FD codes.The parallelization agent detects the problem but it cannot make a correction. parAgent alerts the user by giving a message.

This type of situations occur in the MM5 code mainly due to variable name aliasing used by the programmer (see section 4.1.3). In this particular situation, the user probably implicitly associated the variable b with the j loop index 1. To fix this problem, the user will have to make this association explicit by first converting b to a 3D array with a dimension mapped to the j loop index and then using the value 1 for j index. Thus, L12 should be converted to b[i,1,k] = 0.5*a[i,1,k]. After this change, parAgent automatically inserts a j loop (just like handling of lines L6 to L8). Thus, line L12 is replaced by lines L13 and L14 in the post-diagnostics code.

Finally, consider the fragment L14 to L17. Line L15 violates the Standardization rule that loop indices i and j cannot be modified in any way other than in do loops. parAgent alerts the user to this violation and lets him modify the code. The lines L14 to L17 are replaced by lines L16 to L19 in the post-processed code.

5.3 Analysis Of Communication Requirements And Display Of Stencils

This is one of the most important capabilities of parAgent. For each subroutine, parAgent converts the code into an internal block representation and then performs analysis on the blocks to determine the communication requirements. A sample test code and the corresponding internal representation is shown in Figure 13. As can be seen from the figure, a block encapsulates many lines of code. Each block has its own read and write sets. In this simple test case, the variables b and c written in Block1 are read in Block2 and the variable a written in Block2 is read in Block3.

parAgent also displays the communication requirement findings in form of stencils. Figure 14 shows the block and stencil display for the test code. The display shows Block1 computation is followed by communication of variables b and c. The variables are exchanged according to the pattern shown in the stencils for b and c. This communication is followed by

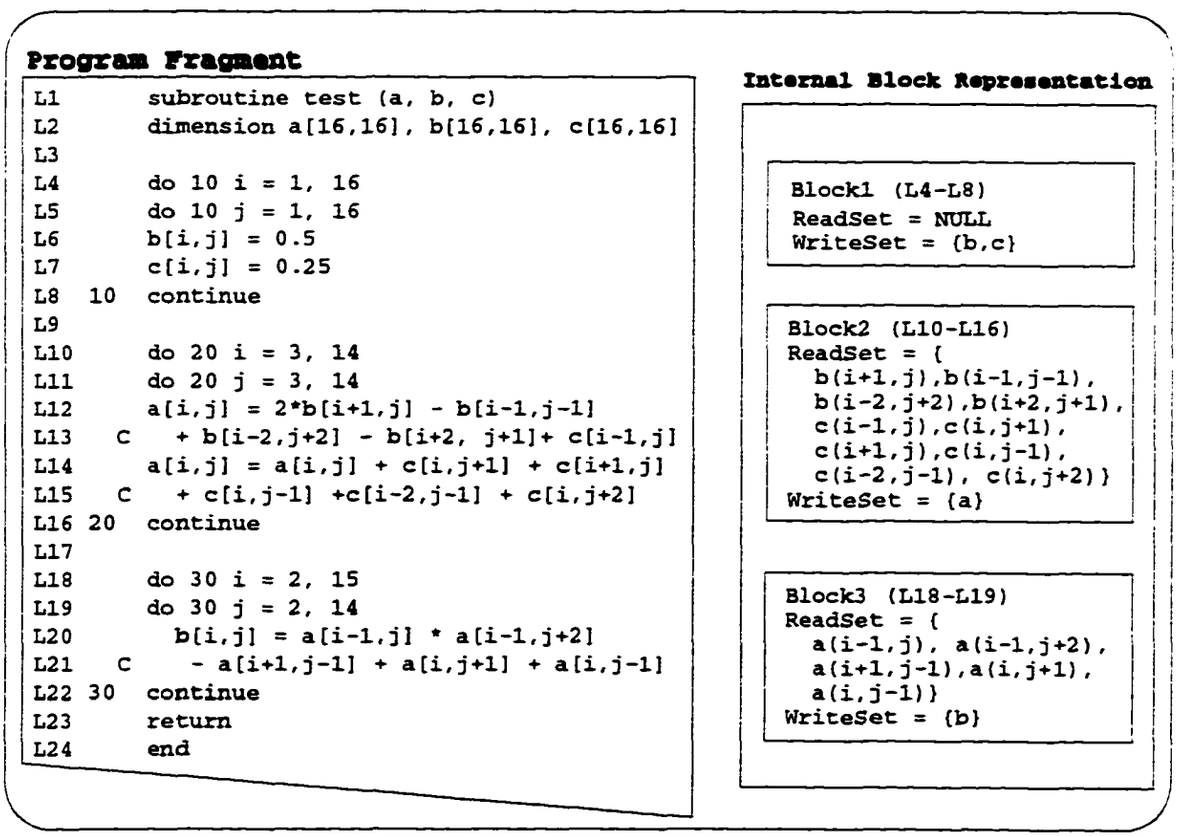


Figure 13: Internal Block

Block2 computation. Next, another communication takes place - this time for variable a. The stencil for variable a shows the communication pattern. This is followed by the Block3 computation. The stencils represent the communication pattern. The square blob represents the current processor which does a read. The round blobs represent the neighboring processors and their relative positions on the grid. These processors do a write.

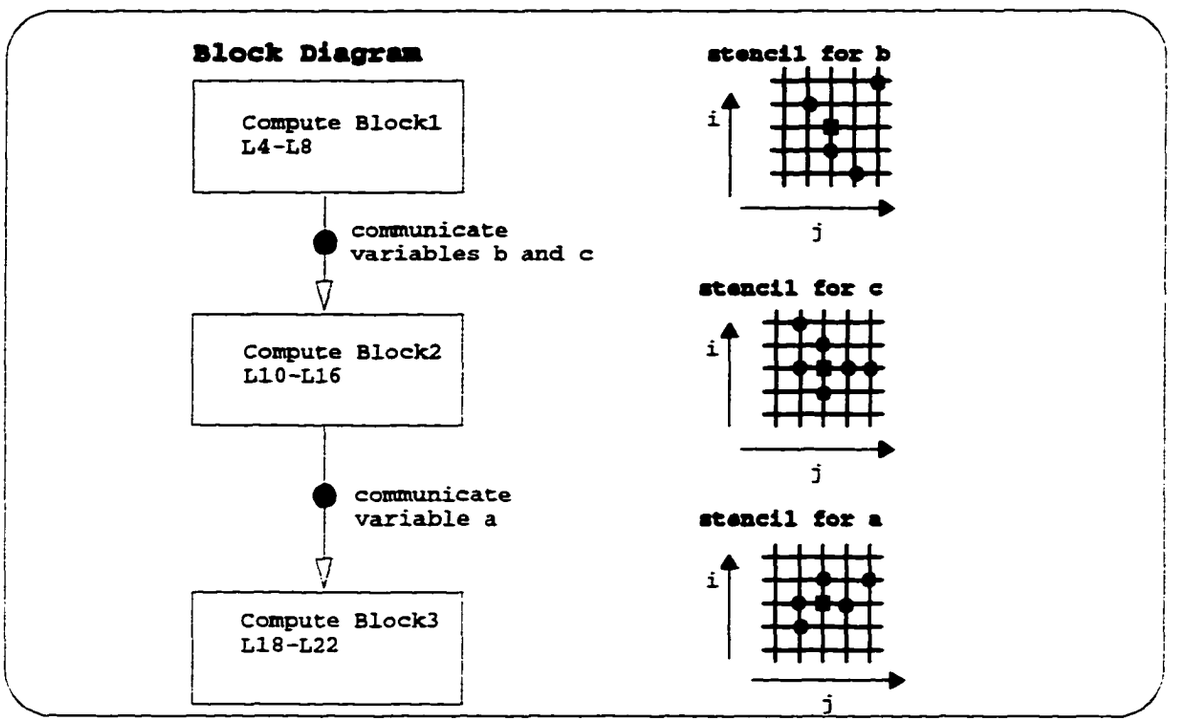


Figure 14: Blocks and Stencils Display

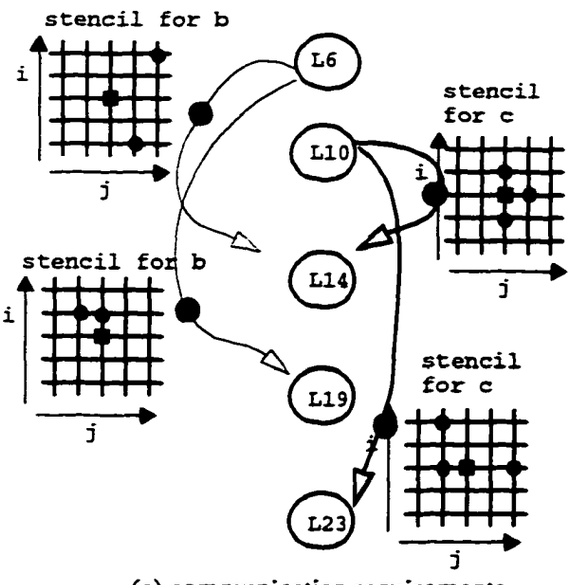
5.4 Communication Optimization

This is another very important capability of parAgent. Communication overhead is a function of the number of times communication takes place. Coalescing messages going to the same processor is a standard way to reduce the communication cost. A large code can contain hundreds of data exchange points.

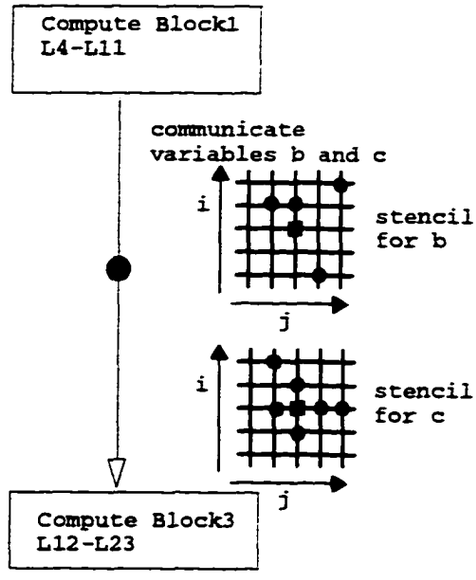
The test code shown in Figure 15 serves to demonstrate parAgent's capability to optimize communication. (a) in Figure 15 shows the communication requirements of the

Program Fragment

```
L1       subroutine test (a, b, c)
L2       dimension a[16,16], b[16,16], c[16,16]
L3
L4       do 10 i = 1, 16
L5       do 10 j = 1, 16
L6   10  b[i,j] = 0.5
L7
L8       do 20 i = 1, 16
L9       do 20 j = 1, 16
L10  20  c[i,j] = 0.25
L11
L12      do 30 i = 3, 14
L13      do 30 j = 3, 14
L14  30  a[i,j] = 2*c[i,j+1] -c[i+1,j]
L15    C   + c[i-1,j] +b[i-2,j+2] -b[i+2,j+1]
L16
L17      do 40 i = 3, 14
L18      do 40 j = 3, 14
L19  40  a[i,j] = a[i,j] +b[i-1,j] +b[i-1,j-1]
L20
L21      do 50 i = 3, 14
L22      do 50 j = 3, 14
L23  50    a[i,j] = c[i,j-1] +c[i-2,j-1] +c[i,j+2]
L24      return
L25      end
```



(a) communication requirements



(b) parAgent blocks and stencils

Figure 15: Communication

code. (b) shows that parAgent obtains only one data exchange point at which the variables b and c are communicated according to the stencils displayed.

As can be seen from the figure, the current processor (depicted by the square blob on the stencil) reads both the variables b and c from the processor immediately above the current processor (p_{up}). During code generation, parAgent makes Runtime System Library (RSL)[26] communicate calls to perform the data exchange and lets RSL pack the messages together. In this particular case, RSL packs the variables b and c into one message and sends it from p_{up} to the current processor.

5.5 Inter-procedural Analysis

This capability is essential because most legacy codes involve several procedures. The test code shown in Figure 16 highlights the issues involved in inter-procedural analysis.

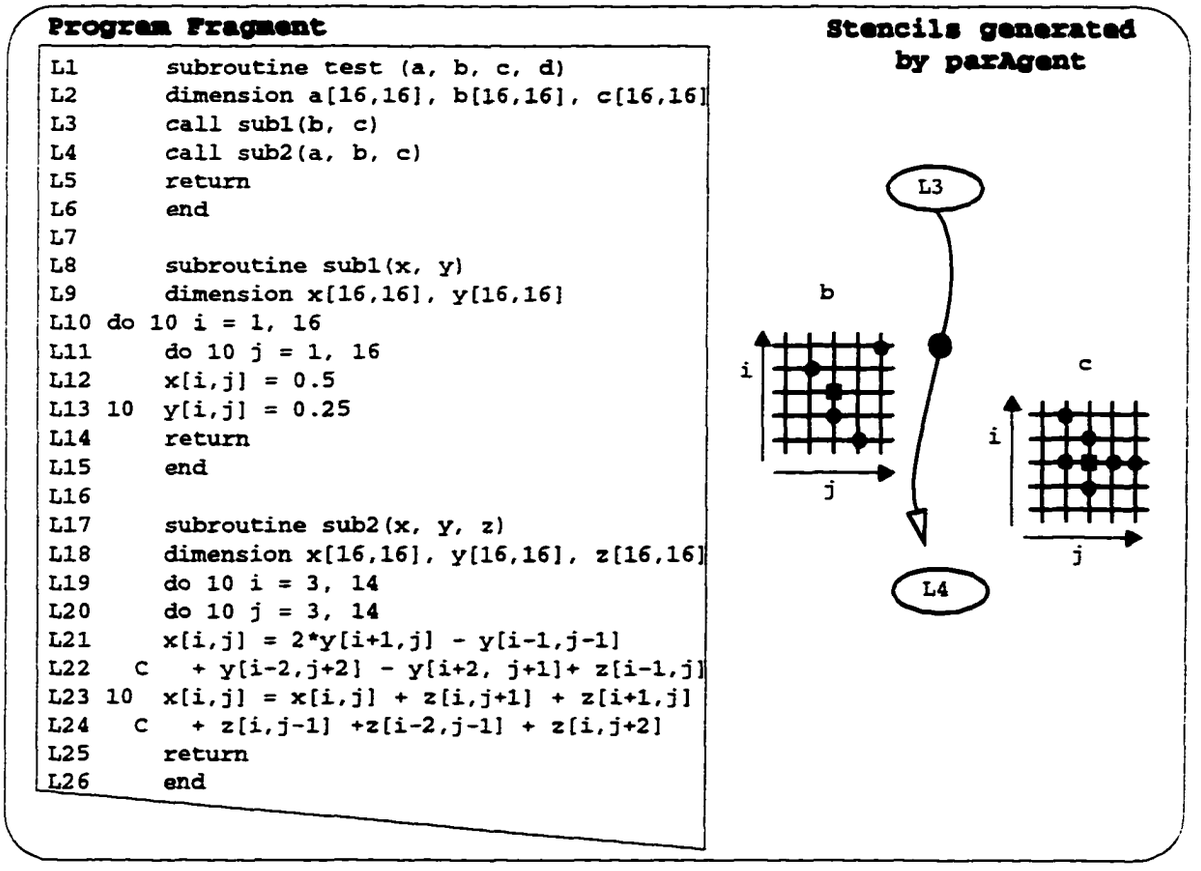


Figure 16: Inter-procedural Analysis

First, parAgent processes sub1 and then sub2 and obtains their read and write sets. As a block, subroutine sub1 has the write set {x,y} and the read set {}. Similarly, as a block subroutine sub2 has the write set {x} and the read set {y(i+1,j), y(i-1,j-1), y(i-2,j+2), y(i+2,j+1), z(i-1,j), z(i,j+1), z(i+1,j), z(i,j-1), z(i-2,j-1), z(i,j+2)}. Note that these sets are in terms of the subroutines formal arguments. parAgent stores information regarding the read and write sets for formal arguments and global variables in a table indexed by the function name.

Next, parAgent starts analysis of subroutine test. When the call sub1 statement is encountered, parAgent attaches the read write sets (which it obtains from the table) of the subroutine to the statement by replacing the formal arguments with the actual parameters being passed to the subroutine. It also attaches the read write sets of the global variables. In this example, L3 is assigned the read set {} and the write set {b, c}. Similarly, L4 is assigned the read set {b(i+1,j), b(i-1,j-1), b(i-2,j+2), b(i+2,j+1), c(i-1,j), c(i,j+1), c(i+1,j), c(i,j-1), c(i-2,j-1), c(i,j+2)} and the write set {a}. During communication analysis of subroutine test, parAgent finds the communication between L3 and L4. The stencils for communication of the variables b and c are shown in the figure.

5.6 Loop Split - Detection And Action

Another capability of parAgent which has a big impact on performance of parallel code is its ability to perform loop split when necessary.

Consider the "before split" test code and the corresponding parallel code as shown in Figure 17. Assume that a 4x4 processor grid is being used. As shown, each processor will execute the communication statement within the i loop (L4-L13). Now consider the "after split" test code shown in Figure 18. Note that this test code is identical to the "before split" test code in functionality. However, the corresponding parallel code executes the communication statement only once.

In explicit finite difference problems, computations proceed at the different grid points followed by a data exchange. Given this model of computation, it is assumed that loop split is always feasible and that the code after the loop split is identical to the code before loop split.

Program Fragment

```
L1        subroutine test (a, b)
L2        dimension a[16,16], b[16,16]
L3
L4        do 10 i = 1, 16
L5        do 20 j = 1, 16
L6   20   b[i,j] = 0.5
L7
L8        do 30 j = 3, 16
L9   30   a[i,j] = b[i,j-1] + b[i, j-2]
L10
L11  10   continue
L12       return
L13       end
```

Parallel code

```
L1        subroutine test (a, b)
L2        dimension a[4,4], b[4,4]
L3
L4        do 10 i = 1, 4
L5        do 20 j = 1, 4
L6   20   b[i,j] = 0.5
L7        COMMUNICATE B
L8        do 30 j = 1, 4
L9        if(jg(j).ge.3) then
L10       a[i,j] = b[i,j-1] + b[i, j-2]
L11       endif
L12  30   continue
L13  10   continue
L14       return
L15       end
```

Figure 17: Before Loop Split

5.7 Parallel Code Generation

After program analysis is completed, the user can choose to generated parallel code for the analyzed files. The current version of parAgent makes use of the RSL library. Generation of parallel code consists of: creation of a driver routine, creation of communication stencils, and modification of the serial code. This example serves to illustrate the parallel code generation process.

Program Fragment after split

```
L1        subroutine test (a, b)
L2        dimension a[16,16], b[16,16]
L3
L4        do 10 i = 1, 16
L5        do 10 j = 1, 16
L6   10   b[i,j] = 0.5
L7
L8        do 20 i = 1, 16
L9        do 20 j = 3, 16
L10  20   a[i,j] = b[i,j-1] + b[i, j-2]
L11
L12       return
L13       end
```

Parallel code

```
L1        subroutine test (a, b)
L2        dimension a[4,4], b[4,4]
L3
L4        do 10 i = 1, 4
L5        do 10 j = 1, 4
L6   10   b[i,j] = 0.5
L7        COMMUNICATE B
L8        do 20 i = 1, 4
L9        do 20 j = 1, 4
L10       if(jg(j).ge.3) then
L11       a[i,j] = b[i,j-1] + b[i, j-2]
L12  20   continue
L13       return
L14       end
```

Figure 18: After Loop Split

Consider the serial code shown in Figure 19. The blocks and stencils for communication, obtained after analysis of the serial code, are shown in Figure 20. There are two communication points. At the first communication point variables b and c are exchanged between the processors as per the stencils shown. At the second communication point, variables a and d are exchanged between the processors as per the stencils shown.

First, the driver routine (See Figure 21) is created. In the driver code, Lines L8 to L12 represents RSL initialization calls. Line L14 represents definition of communication stencils and L16 is call to the parallel test code. The processor array size is specified by the user in line L6. Several elements of an array will map to one processor (processor virtualization) and the mapping function is invoked by the call to rsl_fcn_decompose on Line L11. Note that RSL allows nesting of domains and irregular blocks. This version of parAgent does not make use of these features of RSL.

Next, the communication specification routine is generated (see Figure 22 and Figure 23). Here the stencils for communication are created and compiled as required by RSL.

```
L1       subroutine test
L2       dimension a[16,16], b[16,16]
L3       dimension c[16,16], d[16,16,16]
L4
L5       do 10 i = 1, 16
L6       do 10 j = 1, 16
L7       b[i,j] = 0.5
L8       c[i,j] = 0.25
L9   10  continue
L10
L11      do 20 i = 3, 14
L12      do 20 j = 3, 14
L13      a[i,j] = 2*b[i+1,j] - b[i-1,j-1] + b[i-2,j] - b[i+2, j]+ c[i-1,j]
L14      a[i,j] = a[i,j] + c[i,j+1] + c[i+1,j] + c[i,j-1] +c[i-2,j] + c[i,j+2]
L15      do 20 k = 1,16
L16        d[k,i,j] = 12.1
L17  20  continue
L18
L19      do 30 i = 2, 15
L20      do 30 j = 2, 14
L21        b[i,j] = a[i-1,j] - a[i+1,j-1] + a[i,j+1] + a[i,j-1]
L22        do 30 k = 1, 16
L23          d[k,i,j] = d[k, i-1,j-1]  - 2.0*d[k,i+1,j]
L24  30  continue
L25      return
L26      end
```

Figure 19: The serial code

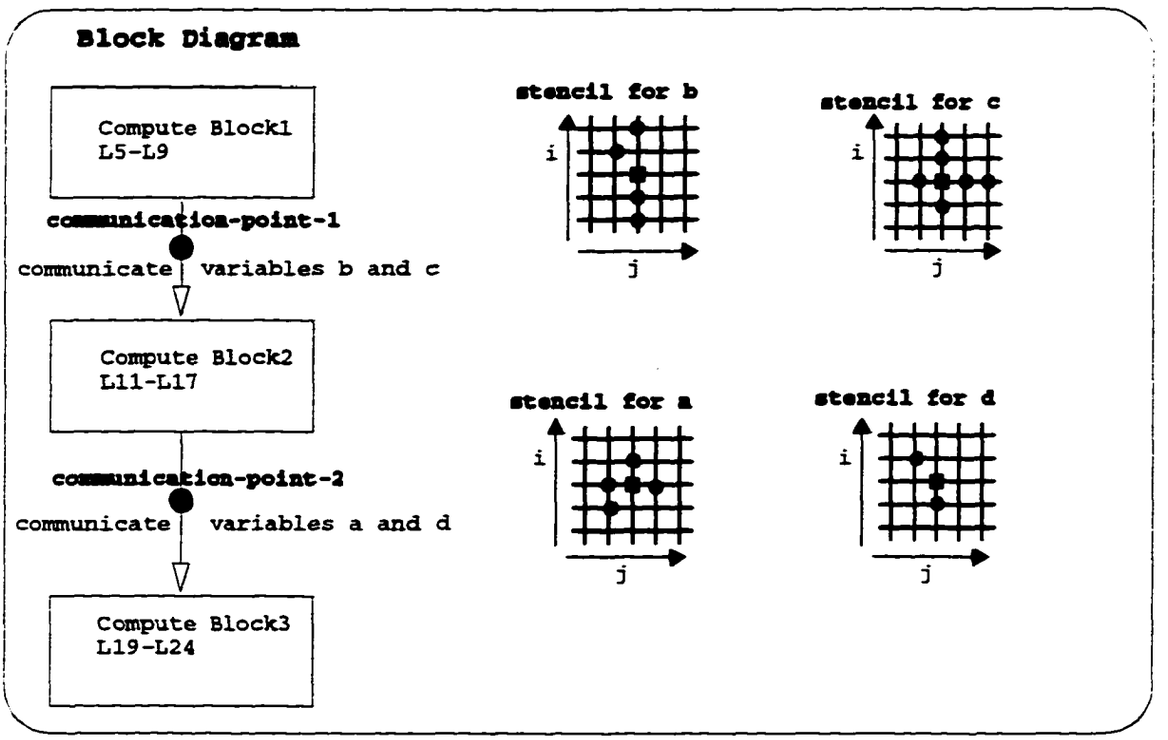


Figure 20: The Blocks and Stencils Display

```
L1      program driver
L2
L3    #  include "rsl.inc"
L4    #  include "rslcom.inc"
L5
L6      read * nproc_lt, nproc_ln   !User specified processor array size
L7
L8      call rsl_initialize (nproc_lt, nproc_ln)
L9
L10     call rsl_mother_domain(domains(1), RSL_12PT, IL, JL, nproc_lt, nproc_ln)
L11     call rsl_fcn_decompose(domains(1), mapping)
L12     call rsl_new_decomposition(domains)
L13
L14     call define_stencil(domains(1))
L15
L16     call test(domains(1))
L17
L18     return
L19     end
```

Figure 21: Driver routine

```
L1      subroutine define_stencils (domain)
L2    # include "rsl.inc"
L3    # include "rslcom.inc"
L4      integer domain
L5      integer dec2dij(2), ll2dij(2), gl2dij(2)
L6      integer dec3dkij(3), ll3dkij(3), gl3dkij(3)
L7
L8      dec2dij(1) = RSL_NORTHSOUTH
L9      dec2dij(2) = RSL_EASTWEST
L10     ll2dij(1) = MIX
L11     ll2dij(2) = MJX
L12     gl2dij(1) = IL
L13     gl2dij(2) = JL
L14
L15     dec3dkij(1) = RSL_NOTDECOMPOSED ! vertical dimension
L16     dec3dkij(2) = RSL_NORTHSOUTH
L17     dec3dkij(3) = RSL_EASTWEST
L18     ll3dkij(1)  = KL
L19     ll3dkij(2)  = MIX
L20     ll3dkij(3)  = MJX
L21     gl3dkij(1)  = KL
L22     gl3dkij(2)  = IL
L23     gl3dkij(3)  = JL
L24
L25   C CREATE STENCILS FOR COMMUNICATE-POINT-1
L26     call rsl_create_message(cp1n2)
L27     call rsl_build_message(cp1n2, RSL_REAL, b, 2, dec2dij, gl2dij, ll2dij)
L28     call rsl_build_message(cp1n2, RSL_REAL, c, 2, dec2dij, gl2dij, ll2dij)
L29
L30     call rsl_create_message(cp1n1)
L31     call rsl_build_message(cp1n1, RSL_REAL, c, 2, dec2dij, gl2dij, ll2dij)
L32
L33     call rsl_create_message(cp1nw)
L34     call rsl_build_message(cp1nw, RSL_REAL, b, 2, dec2dij, gl2dij, ll2dij)
L35
L36     call rsl_create_message(cp1w1)
L37     call rsl_create_message(cp1e1)
L38     call rsl_create_message(cp1e2)
L39     cp1w1 = cp1n1
L40     cp1e1 = cp1n1
L41     cp1e2 = cp1n1
L42
L43     call rsl_create_message(cp1s1)
L44     cp1s1 = cp1n2
L45
L46     call rsl_create_message(cp1s2)
L47     cp1s2 = cp1nw
L48
L49
```

Figure 22: Stencil Initialization code

```
L50
L51     cp1messages(1)  =                       cp1n2
L52     cp1messages(2)  =           cp1nw
L53     cp1messages(3)  =                       cp1n1
L54     cp1messages(4)  =                                   RSL_INVALID
L55     cp1messages(5)  = RSL_INVALID
L56     cp1messages(6)  =            cp1w1
L57     cp1messages(7)  =                                   cp1e1
L58     cp1messages(8)  =                                            cp1e2
L59     cp1messages(9)  =         RSL_INVALID
L60     cp1messages(10) =                       cp1s1
L61     cp1messages(11) =                                   RSL_INVALID
L62     cp1messages(12) =                       cp1s2
L63
L64     call rsl_create_stencil(cp1sten)
L65     call rsl_describe_stencil(cp1sten, RSL_12PT, cp1messages)
L66
L67
L68 C   CREATE STENCILS FOR COMMUNICATION-POINT-2
L69     call rsl_create_message(cp2nw)
L70     call rsl_create_message(cp2s1)
L71     call rsl_build_message(cp2nw, RSL_REAL, d, 3, dec3dkij, gl3dkij,ll3dkij)
L72     cp2s1 = cp2nw
L73
L74     call rsl_create_message(cp2n1)
L75     call rsl_create_message(cp2w1)
L76     call rsl_create_message(cp2e1)
L77     call rsl_create_message(cp2sw)
L78     call rsl_build_message(cp2n1, RSL_REAL, a, 2, dec2dij, gl2dij, ll2dij)
L79     cp2w1 = cp2n1
L80     cp2e1 = cp2n1
L81     cp2sw = cp2n1
L82
L83     cp2messages(1)  =           cp2nw
L84     cp2messages(2)  =                 cp2n1
L85     cp2messages(3)  =                       RSL_INVALID
L86     cp2messages(4)  =           cp2w1
L87     cp2messages(5)  =                       cp2e1
L88     cp2messages(6)  =           cp2sw
L89     cp2messages(7)  =                 cp2s1
L90     cp2messages(8)  =                        RSL_INVALID
L91
L92     call rsl_create_stencil(cp2sten)
L93     call rsl_describe_stencil(cp2sten, RSL_8PT, cp2messages)
L94
L95     call rsl_compile_stencil(domain, cp1sten)
L96     call rsl_compile_stencil(domain, cp2sten)
L97     return
L98     end
```

Figure 23: Stencil initialization code (continued)

Messages are created and built up by the rsl_create_message and rsl_build_message (See lines L26-49). Note that all variables directed towards the same processor are built into the same message. All messages at a communication point are described in terms of a 8pt, 12pt, or 24pt stencil (See lines L51 - L65). In this example, two stencils cp1sten and cp2sten are created corresponding to the two communication points. The stencils must be compiled by the rsl_compile_stencil (See line L95-L96) before they can be used. RSL takes care of packing of messages and unpacking of messages and delivery of messages according to the stencil.

Finally, the serial code is converted to parallel code (See Figure 24). Several modifications to the code occur. First, the arrays are partitioned among the processors. The local arrays are padded on all sides as required by RSL communication routines. The local areas are declared with their new sizes (Lines L3-L6). Second, a call is made to rsl_get_run_info to obtain loop index information required by RSL. Third, each j loop is replaced by RSL_MAJOR_LOOP(j) and RSL_END_MAJOR_LOOP constructs. Also, each i loop is replaced by RSL_MINOR_LOOP(i) and RSL_END_MINOR_LOOP constructs. In the case that the original loops did not run over the full dimension of the i dimension or j dimension, RSL_BOUND and RSL_END_BOUND statements are entered as necessary. Finally, call rsl_exch_stencil statements are introduced at the two communication points. Note that the stencils cp1sten and cp2sten are used to describe the communication requirement to the rsl_exch_stencil routine. The parallel code, the driver routine, and the communication routine must be compiled and linked with the RSL library and is then ready to run.

```
L1  #    include "LoopMacros.inc"
L1       subroutine test (domain)
L2       integer domain
L3       dimension a[16/nproc_lt +2*PADAREA,16/nproc_ln +2*PADAREA]
L4       dimension b[16/nproc_lt +2*PADAREA,16/nproc_ln +2*PADAREA]
L5       dimension c[16/nproc_lt +2*PADAREA,16/nproc_ln +2*PADAREA]
L6       dimension d[16, 16/nproc_lt +2*PADAREA,16/nproc_ln +2*PADAREA]
L7       integer maxruns
L8       parameter (maxruns = JL)
L9       RSL_DECLARE_RUN_VARS(maxruns)
L10      call rsl_get_run_info(domains, maxruns, RSL_RUN_ARGS)
L11
L12      RSL_MAJOR_LOOP(j)
L13      RSL_MINOR_LOOP(i)
L14      b[i,j] = 0.5
L15      c[i,j] = 0.25
L16      RSL_END_MINOR_LOOP
L17      RSL_END_MAJOR_LOOP
L18
L19      call rsl_exch_stencil(domain, cp1sten)
L20      RSL_MAJOR_LOOP(j)
L21      RSL_BOUND(j, 3, 14)
L22      RSL_MINOR_LOOP(i)
L23      RSL_BOUND(i, 3, 14)
L24      a[i,j] = 2*b[i+1,j] - b[i-1,j-1] + b[i-2,j] - b[i+2, j]+ c[i-1,j]
L25      a[i,j] = a[i,j] + c[i,j+1] + c[i+1,j] + c[i,j-1] +c[i-2,j] + c[i,j+2]
L26      do 20 k = 1,16
L27        d[k,i,j] = 12.1
L28 20   continue
L29      RSL_END_BOUND
L30      RSL_END_MINOR_LOOP
L31      RSL_END_BOUND
L32      RSL_END_MAJOR_LOOP
L33
L34      call rsl_exch_stencil(domain, cp2sten)
L35      RSL_MAJOR_LOOP(j)
L36      RSL_BOUND(j, 2, 14)
L37      RSL_MINOR_LOOP(i)
L38      RSL_BOUND(i, 2, 14)
L39        b[i,j] = a[i-1,j] - a[i+1,j-1] + a[i,j+1] + a[i,j-1]
L40        do 30 k = 1, 16
L41          d[k,i,j] = d[k, i-1,j-1]  - 2.0*d[k,i+1,j]
L42 30     continue
L43      RSL_END_BOUND
L44      RSL_END_MINOR_LOOP
L45      RSL_END_BOUND
L46      RSL_END_MAJOR_LOOP
L47      return
L48      end
```

Figure 24: The parallel code

CHAPTER 6 SUMMARY

Making existing legacy codes ready for parallel processing poses various challenges. These codes are very large and complex and manual parallelization has proven to be extremely time-consuming and error-prone. Furthermore, existing parallelization tools cannot handle the complexity of these legacy codes and are unable to parallelize them.

Using a totally new approach, we have developed parAgent, a tool to parallelize codes based on the explicit FD method. Our approach focuses on special classes of codes as opposed to parallelization of arbitrary codes. The advantage is that we are able to use high-level knowledge of the special class to manage the complexity of the parallelization problem. A blend of automation and user assistance is used to provide a pragmatic solution for parallelization of specific classes of codes: it requires the user to specify the high-level knowledge, but automates tasks which are time-consuming, tedious, and error-prone.

As explained in the previous chapter, the working of parAgent has been verified by running against a test-suite designed to test the individual features of the tool. A sequential code developed to contain the complexities commonly faced during parallelization of legacy codes has been parallelized and run on a parallel computer. Inspection of the parallel code confirms that parAgent produces efficient parallel code.

More excitingly, parAgent has now been used on MM5, RADM, and RAMS. parAgent has been demonstrated to researchers at different Institutes: NASA Ames, California; CDAC, India; EPA, North Carolina; University of Athens, Greece; Visitor from NCAR; Meteorological Institute, Korea; System Engineering Research Institute, Korea; and Astor Corporation, Colorado. In these codes, there is an initialization code fragment preceding the FD code fragment. This initialization code fragment does not belong to FD class. Work is in progress on parallelization of the initialization code. However, all the other stages of parallelization including inter-procedural analysis and generation of communication have been completed and verified. Qualitatively, the parallelization have been found to be equivalent to manual parallelization in performance. Amazingly, it took only a few weeks to parallelize each of these legacy codes.

This new approach can be applied to a variety of problem domains and it has the potential to enrich the development of parallel computing. Considerable work is needed if one intends to cover each class in detail. Each one of these numerical methods allow multiple variations and in reality one thinks of many subclasses hidden inside each class. Overall, we believe that the research will play an important part to accelerate the evolution of high performance technologies and their applications to scientific and engineering problems. The key benefits are: substantial reuse of existing software and considerable saving of time and effort for developing efficient parallel code.

Although the new approach and parAgent have been developed with parallelization as the main objective, the information provided by the tool can be used for various purposes. For example, the communication stencil display provides useful information to the application scientist about the underlying numerical method and the exchange of data. Similarly, the block representation of the code based on the high level knowledge of a specific class is useful to reason about the sequential or parallel code. Also, information such as the call-tree, the indexing scheme for variables, and the control-flow structure can be used to examine existing legacy codes.

APPENDIX A FD RULES

FD Class Rules

1. Only near-neighbor data-transfer.
2. Scalars(includes non-idx arrays) initize in loop before use. Also, scalars are not used later - or are re-initialzed before use elsewhere.
3. Inside IF's with A[i,j] there cannot be commn.
4. No commn within an i/j loop
5. In a [i [j C1] [j C2]] loop, the following dependencies are NOT allowed
   1 if array A write in C1, then C2 not have A[i+k,...] reads
   2 if array A write in C2, then C1 not have A[i-k,...] reads

Standardization Rules

1. Consistent use of program indices to refer to spatial indices(eg i,j mean NS and EW directions always).
2. IF's with i or j indexes must be cascading (to allow processing) because r/w's of then/else parts can interfere
3. i,j positions in subscript for arrays remain fixed
4. Arrays access by i+-k1, j+-k2 ; k1 and k2 known constants.
5. Array accessed by specific subscripts not allowed
6. Only writes to [i,j] subscripted arrays
7. i and j written to only as DO i= 1, 16 etc, not using assignment statement.
8. Arrays are not aliased to lower dimensions. ex: array with 2D is not used like a 1D array
9. Loops with i or j indexes have constants for Upper and Lower bounds.
10. Structured program (i.e. no gotos,implied ifs etc).
11. variables shall not be aliased in COMMON and EQUIVALENCE statements; they shall not overlap with each other.

REFERENCES

[1] ADAPTOR, http://www.gmd.de/SCAI/lab/adaptor/adaptor_home.html, Dr. Thomas Brandes, accessed on 7 Nov 1997.

[2] Ahmed, N. Carriero, and D. Gelernter, The Linda program builder, in Third Workshop Languages and compilers for parallelism, MIT Press, Cambridge, MA, 1991.

[3] J. M. Anderson and M.S. Lam, Global optimizations for parallelism and locality on scalable parallel machines, ACM SIGPLAN Notices, 28 (1993), pp. 112-125.

[4] R. A. Anthes and T.T. Warner, Development of hydrodynamic models suitable for air pollution and other mesometeorological studiews, Mon. Weather review, 106 (1978), pp. 1045-1078.

[5] D.F. Bacon, S.L. Graham, and O.J. Sharp, Compiler transformations for high performance computing, JACS, 26 (1994), pp. 345-420.

[6] Banatre and D.L. Metayer, The Gamma Model and its discipline of programming, Science of Computer Programming, 15 (1990), pp. 55-77.

[7] M. Chandy and J. Misra, "Parallel program design: A foundation," Addison-Wesley, Reading, MA, 1988.

[8] D.Y. Cheng, A Survey of parallel programming languages and tools, Tech. Rep. RND-93-005, NASA Ames Research Center, Moffet Field, CA, 1993.

[9] Culler, D.E. Karp, R.M. Patterson, D.A. Sahay, A. Schauser, K.E. Santos, E. Subramanian, and Von Eicken, "LogP: Towards a realistic model of parallel computing", Fourth ACM SIGPLAN symposium on principles and practice of parallel programming, May 1993.

[10] D System project, http://www.cs.rice.edu/~dsystem, Fortran Parallel Programming Systems group, accessed on 7 Nov 1997.

[11] J.J. Dongorra and B. Tourancheau, Environments and tools for parallel scientific computing, Advances in Parallel Computing, Vol. 6, Elvesier Science Publishers BV, (North-Holland),1993

[12] FORGExplorer, http://www.apri.com, Applied Parallel Research Inc., accessed on 7 Nov 1997.

[13] Fortran 90D compiler, http://www.npac.syr.edu/users/haupt/f90d/compilerhome.html, T. Haupt, accessed on 7 Nov 1997.

[14] Fx project, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/iwarp/member/fx/public/www/fx.html, The Fx project group, accessed on 7 Nov 1997.

[15] M.P.I. Forum, Document for a standard message passing interface, Tech. Rep. CS-93-214, University of Tennessee, TN, 1994.

[16] R. Freidman, J. Levesque, and G. Wagenbreth, Fortran Parallelization Handbook, Applied Parallel Research, 1995.

[17] Gannon, F. Bodin, S. Srinivas, N. Sundaresan, and S. Narayan, "Sage++: An object oriented toolkit for program transformations," Tech. Rep., Dept. of Computer Science, Indiana Univeristy, 1993

[18] A. Goldberg, P. Mills, L. Nyland, J. Prins, J. Reif, and J. Riely, Specification and Development of parallel algorithms with the Proteus system, DIMACS, 18 (1994), pp. 383-399.

[19] R.E. Griswold, The ICON programming language, Prentice Hall, Upper Saddle River, NJ,1990.

[20] S. Hiranandani, K. Kennedy, C.W. Tseng, and S. Warren, The D editor: A new interactive parallel programming tool, in proceedings of Supercomputing conference, 1994, pp.733-742.

[21] S. Kothari, H. Oh, and E. Gannet, "Optimal Designs of Linear-flow Systolic Architectures," International Conference of Parallel Processing, 1989, pp.247-256.

[22] J. Li and M. Chen, The data alignment phase in compiling programs for distributed memory machines, Journal of Parallel and Distributed Computing, 13 (1991), pp.213-221.

[23] M. Mace, Memory storage patterns in parallel processing, Kluwer Academic, Boston, M.A, 1987.

[24] B. Massingill, Mesh computations, Obtained via http://www.etext.caltech.edu, June 1995.

[25] P. Messina and T. Sterling, System Software and Tools for high performance computing Environments, SIAM publication, Philadelphia, PA,1993.

[26] J. Michalakes, RSL: A parallel runtime system library for regular grid finite difference models using multiple nests, Tech. Rep. ANL/MCS-TM-197, MCS Division, Argonne National Laboratory, Argonne, IL, 1994.

[27] J. Michalakas, T. Canfield, R. Nanjundiah, and S. Hammond, Parallel implementation, validation, and performance of MM5, in Proc. 6th Workshop on the use of Parallel processors in Meteorology, Reading, U.K., 1994, European Center for Medium Range Weather Forecasting.

[28] PARADIGM project, http://www.crhc.uiuc.edu/Paradigm, Center for Reliable and High-Performance Computing, accessed on 7 Nov 1997.

[29] parAgent, http://www.cs.iastate.edu/~hpc/paragent.html, Aravind Krishnaswamy, accessed on 1 Dec 1997.

[30] PGI HPF compiler, http://www.pgroup.com, Portland Group Inc, accessed on 7 Nov 1997.

[31] POLARIS, http://polaris.cs.uiuc.edu/polaris/polaris.html, The Polaris Compiler Group, accessed on 7 Nov 1997.

[32] Charles Rich and Richard C. Walters, The programmer's Apprentice, ACM Press, New York, NY, 1990.

[33] sHPF compiler, http://www.ccg.ecs.soton.ac.uk/Projects/shpf/shpf.html, John Merlin, accessed on 7 Nov 1997.

[34] Skillicorn, Architecture independent parallel computation, IEEE computer, 23 (1990), pp. 38-51.

[35] D.R. Smith, G.B. Kotik, and S.J. Westfold, "Research on knowledge-based software environments at Kestrel Institute," IEEE transactions on Software Engineering, 11 (1985), pp. 1278-1295.

[36] SUIF compiler System, http://suif.stanford.edu/index.html, The Stanford SUIF compiler group, accessed on 7 Nov 1997.

[37] L. Synder, A practical parallel programming model, DIMACS, 18 (1994), pp. 143-160.

[38] E. Soloway and K. Ehrich, "Empirical Studies of Programming Knowledge," IEEE transactions on Software Engineering, 10 (1984), pp. 595-609.

[39] L.G. Valiant, A bridging model for parallel computation, CACM, 8 (1990), pp. 103-111.

[40] Vienna Fortran Compilation System, http://www.par.univie.ac.at/inst/3J/node15.html, Bernd Wender, accessed on 7 Nov 1997.

[41] R.D. Williams, Dime: A programming environment for unstructured triangular meshes on a distributed memory parallel processor, in The third conference on hypercube concurrent computers and applications, Vol 2, 1988, pp. 1770-1787.

[42] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, Reading, MA,1994.

[43] M.Wolfe, "Further Reading in High Performance Compilers," http://www.pgroup.com/~mwolfe/book/further.ps, accessed on 7 Nov 1997.

[44] Zima, SUPERB, University of Vienna, 1989